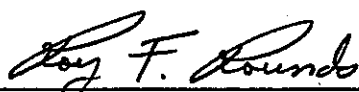


STATE OF CALIFORNIA
DEPARTMENT OF TRANSPORTATION
DIVISION OF NEW TECHNOLOGY,
MATERIALS AND RESEARCH
OFFICE OF ELECTRICAL AND ELECTRONICS ENGINEERING


DEVELOPMENT OF AN ADVANCED
TRANSPORTATION CONTROL COMPUTER

Report Number FHWA/CA/TL-94-08
Caltrans Study Number F90TL12

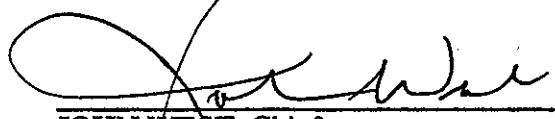
Supervised by..... L.G. Kubel, P.E.
Principal Investigator R.F. Rounds
Co-investigator T.F. Quinlan
Prepared by T.F. Quinlan



ROY ROUNDS, Chief
Electronic Systems
Design and Research Branch



LES KUBEL, Chief
Office of Electrical
and Electronics Engineering



JOHN WEST, Chief
Division of New Technology,
Materials and Research

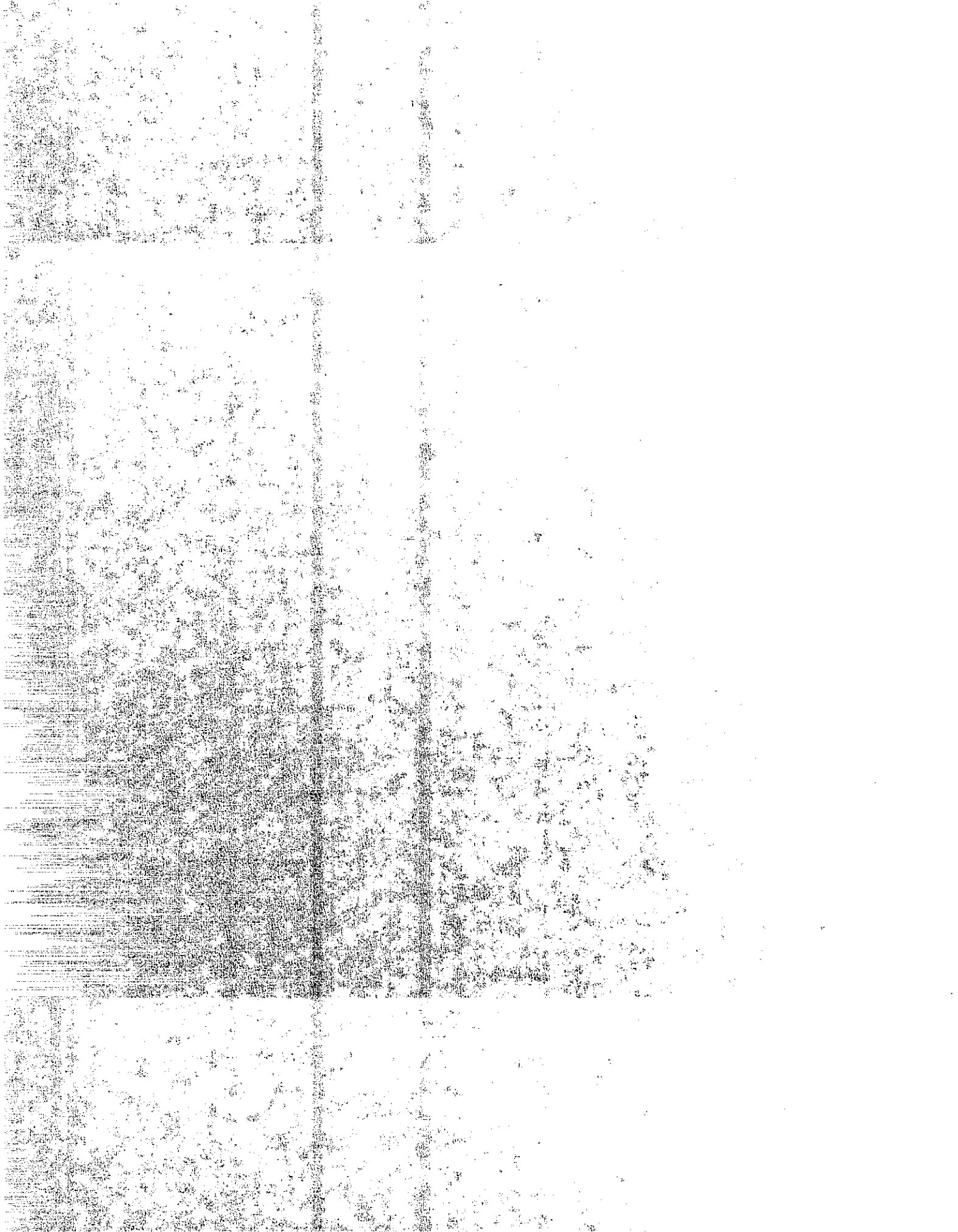
TECHNICAL REPORT STANDARD TITLE PAGE

1. Report No. TL-94-08	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle DEVELOPMENT OF AN ADVANCED TRANSPORTATION CONTROL COMPUTER		5. Report Date June 1993
		6. Performing Organization Code
7. Authors L.G. Kubel, R. F. Rounds, T.F. Quinlan		8. Performing Organization Report No. 631154
9. Performing Organization Name and Address Division of New Technology, Materials and Research California Department of Transportation Sacramento, CA 95819		10. Work Unit No.
		11. Contract or Grant No. F90TL12
12. Sponsoring Agency Name and Address California Department of Transportation Sacramento, CA 95807		13. Type of Report and Period Covered Final
		14. Sponsoring Agency Code
15. Supplementary Notes This project was performed in cooperation with the U. S. Department of Transportation, Federal Highway Administration.		
16. Abstract Advanced hardware and software will be required for successful implementation of Intelligent Vehicle Highway Systems (IVHS). This report presents the development and evaluation of an Advanced Transportation Control Computer (ATC) serving as the basis for standard hardware and software specifications. The ATC prototype employs current engineering technology including an IEEE (Institute of Electrical and Electronic Engineers) standard VME bus, a Motorola 68020 32-bit microprocessor, C high-level programming language, and OS9TM real-time operating system. Controller capabilities were demonstrated in a slow speed weigh-in-motion (SWIM) field test environment. A screening system consisting of the controller, axle sensors, weigh pads, and an inductive loop, determined axle spacings, weights, and classes of trucks entering a static weigh station. The SWIM system alerted weigh station operators to vehicles violating Federal Highway Administration (FHWA) weight/class limits. ATC SWIM system hardware and software was evaluated for accuracy, reliability, and ease of use. Test results found the electrical, mechanical, and environmental performance appropriate for control systems in harsh transportation environments. SWIM system axle spacing measurements fell well within California Department of Transportation accuracy specifications. Weight errors however, were slightly greater than allowed by the Department. Many factors, including pavement irregularities, vehicle suspension dynamics, and SWIM site geometry were believed to have contributed to weight inaccuracies.		
17. Key Words Traffic Control Devices, Traffic Control Computer, Standard Controller, Field Controller, Transportation Controller, ATC		18. Distribution Statement No restriction. This document is available to the public through the National Tech Infor Service, Springfield, VA 2215
19. Security Clasif. (of this report) Unclassified	20. Security Clasif. (of this page) Unclassified	21. No. of Pages

CONVERSION FACTORS

English to Metric System (SI) of Measurement

<u>Quality</u>	<u>English Unit</u>	<u>Multiply By</u>	<u>To Get Metric Equivalent</u>
Length	inches (in) or (")	25.40 .02540	millimetres (mm) metres (m)
	feet (ft) or (')	.3048	metres (m)
	miles (mi)	1.609	kilometres (km)
Area	square inches (in ²)	6.432 x 10 ⁻⁴	square metres (m ²)
	square feet (ft ²)	.09290	square metres (m ²)
	acres	.4047	hectares (ha)
Volume	gallons (gal)	3.785	litre (l)
	cubic feet (ft ³)	.02832	cubic metres (m ³)
	cubic yards (yd ³)	.7646	cubic metres (m ³)
Volume/Time (Flow)	cubic feet per second (ft ³ /s)	28.317	litres per second (l/s)
	gallons per minute (gal/min)	.06309	litres per second (l/s)
Mass	pounds (lb)	.4536	kilograms (kg)
Velocity	miles per hour (mph)	.4470	metres per second (m/s)
	feet per second (fps)	.3048	metres per second (m/s)
Acceleration	feet per second squared (ft/s ²)	.3048	metres per second squared (m/s ²)
	acceleration due to force of gravity (G)	9.807	metres per second squared (m/s ²)
Density	(lb/ft ³)	16.02	kilograms per cubic metre (kg/m ³)
Force	pounds (lb)	4.448	newtons (N)
	kips (1000 lb)	4448	newtons (N)
Thermal Energy	British thermal unit (Btu)	1055	joules (J)
Mechanical Energy	foot-pounds (ft-lb)	1.356	joules (J)
	foot-kips (ft-k)	1356	joules (J)
Bending Moment or Torque	inch-pounds (in-lb)	.1130	newton metres (Nm)
	foot-pounds (ft-lb)	1.356	newton-metres (Nm)
Pressure	pounds per square inch (psi)	6895	pascals (Pa)
	pounds per square foot (psf)	47.88	pascals (Pa)
Plane Angle	degrees (°)	0.0175	radians (rad)
Temperature	degrees fahrenheit (°)	$\frac{^{\circ}\text{F} - 32}{1.8} = ^{\circ}\text{C}$	degrees celsius (°C)
Concentration	parts per million (ppm)	1	milligrams per kilogram (mg/kg)



NOTICE

The contents of this report reflect the views of the Division of New Technology, Materials and Research which is responsible for the facts and the accuracy of the data presented herein. The contents do not necessarily reflect the official views or policies of the State of California or the Federal Highway Administration. This report does not constitute a standard, specification, or regulation.

Neither the State of California nor the United States Government endorse products or manufacturers. Trade or manufacturers names appear herein only because they are considered essential to the object of this document.

ACKNOWLEDGMENTS

This research was sponsored by the United States Department of Transportation, Federal Highway Administration. Special appreciation is due to the following staff members of the Division of New Technology, Materials and Research, Headquarters Division of Traffic Operations, Headquarters Division of Structures, and District 03 Maintenance for their expert assistance: *Weigh-In-Motion System Installation* - Site preparation, material procurement, interface hardware assembly, and construction: Bob Caudle, Bob Cullen, Les Ballinger, Dave Biggs, Eric Jacobson, Jim Anderson, Alan Benson, Ting Co, and Jesse Sandhu. *Electrical Instrumentation* - Test preparation, data acquisition hardware and software: Kai Leung, Wayne Tran, Koney Archuletta, and Rich Quinley. *Asphalt Work and Test Vehicle Operation*: Mark Hills, and Martha Cuevas. *Metalwork*: Gene Wevel, Gene Weldon, Bill Poroshin, and Jesse Perez. *Drafting Services*: Eddie Fong and Irma Gamarra-Remmen. *Logistics and Clerical Services*: Diana Mastroianni and Diane Scherbenske. *California Highway Patrol*: Sergeant Jim Elliot.

TABLE OF CONTENTS

1. Introduction	1
1.1 Problem	1
1.2 Objective	1
1.3 Background	1
2. Conclusion	4
2.1 Prototype Performance	4
2.1.1 ATC Hardware and Software	4
2.1.2 SWIM System	7
2.2 Recommendations	7
2.2.1 ATC Hardware and Software	7
2.2.2 SWIM System	9
3. Implementation	10
4. Technical Discussion	12
4.1 ATC Prototype	12
4.1.1 Hardware	12
4.1.1.1 Backplane	12
4.1.1.2 Microprocessor	18
4.1.1.3 Input/Output and Communications	20
4.1.1.4 Support Hardware	25
4.1.2 Software	29
4.1.2.1 Programming Language	29
4.1.2.2 Operating System	30
4.1.2.3 Object-Oriented Application Development	31
4.1.3 Cost	33

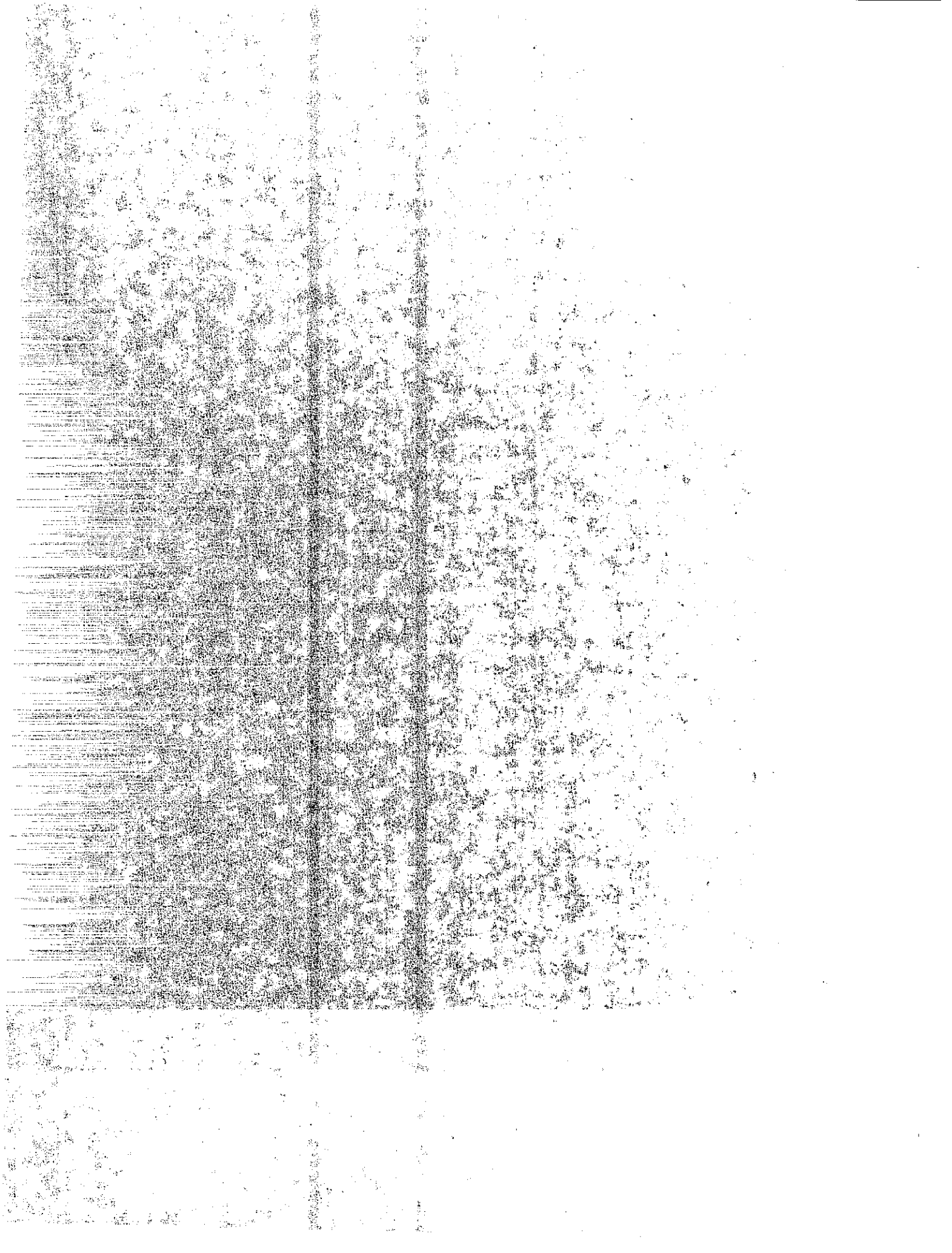
Much of the SWIM application software was written in C. A Microsoft C test routine was developed to simulate a variety of digital axle sensor and analog weigh pad signals representing 15 different vehicle classes. All user interface and computational routines, weight, class, speed, and violation, were also developed in Microsoft C, and later ported to the VME bus ATC platform under OS-9TM C. Very little assembly code was required throughout the project with only a small portion needed for bit-level manipulation of simulation hardware and device drivers. Section 4.2.4 Operational Test, page 60, provides a description of the application software. A complete listing of the test routine and application code is found in Appendix D.

4.1.2.2 Operating System

OS-9TM, selected as the ATC real time operating system, is well suited for many transportation control applications. Run time kernels are small, efficient, and economical. Essential real time elements, such as preemptive task switching and reentrant and position-independent memory modules, allow for execution of a variety of interrupt driven functions with variable frequencies. OS-9TM includes independent file managers for many types of I/O, a fully ROMable kernel, development tools, and a multi-user environment promoting parallel software development. The ATC prototype field controller employs Industrial 68020/OS-9TM, a small real-time kernel designed for ROM-based applications requiring no disk or tape support. The prototype development system required Professional 68202/OS-9TM providing a programming environment with disk and tape support, a C compiler, an assembler, and an assembly debugger. Both Industrial and Professional packages are optimized for the 68020 CPU, while supporting 68000 software development. Additional support tools were purchased to aid in software development including: a system state debugger, a user state debugger, a C source level debugger, PCBRIDGETM a cross development tool supporting MS-DOSTM or OS-2TM based applications, assemblers, linkers, and a communication package.

LIST OF FIGURES

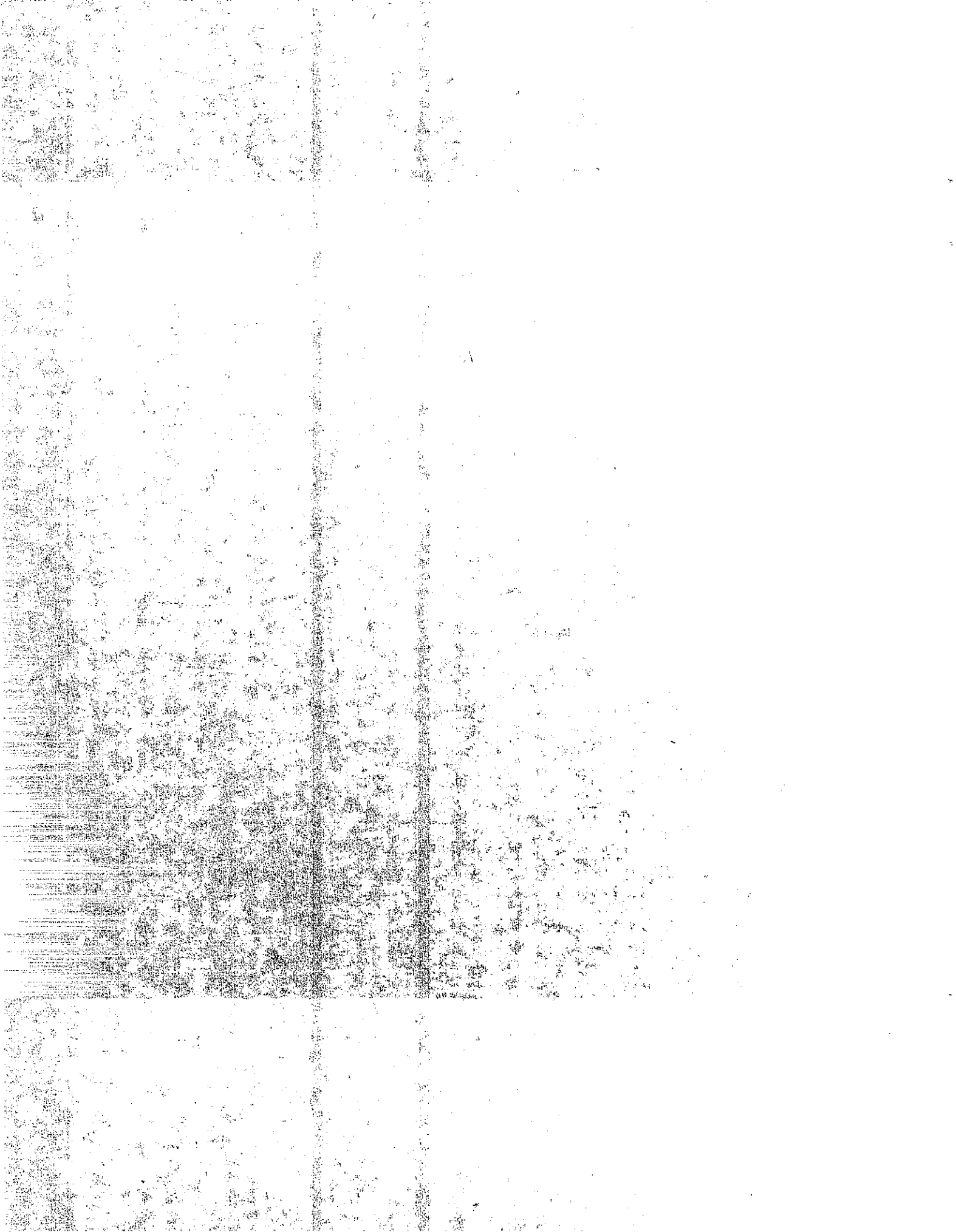
Figure 1.	Functional Modules and Buses Defined by VME BUS IEEE Standard	13
Figure 2.	P1 and P2 VME BUS Connectors	16
Figure 3.	VME BUS Subrack with Mixed Board Sizes	16
Figure 4.	Prototype ATC 6U Development Machine (bottom) and 3U Field Controller (top)	17
Figure 5.	ATC Prototype 3U Field Controller Backplane and Chassis	17
Figure 6.	Functional Block Diagram of ATC Prototype CPU Module	19
Figure 7.	ATC Prototype CPU Module	19
Figure 8.	Functional Block Diagram of ATC Prototype I/O Motherboard Module	21
Figure 9.	Functional Block Diagram of ATC Prototype Digital Input Piggyback Cards	22
Figure 10.	ATC Prototype I/O Motherboard Module and Piggybacks	22
Figure 11.	Functional Block Diagram of ATC Prototype A/D Module	24
Figure 12.	ATC Prototype 3U VME Field Controller	26
Figure 13.	ATC Prototype 3U VME Field Controller with Modular Power Supply	26
Figure 14.	Removable ATC Prototype Power Supply	27
Figure 15.	ATC Prototype Power Supply Back Panel and Connectors	27
Figure 16.	Functional Block Diagram of ATC Prototype Mass Storage Controller Module	28
Figure 17.	Object Oriented ATC Software Operational Concept	32
Figure 18.	Sacramento Area Map with I-80 / Antelope Road Weigh Station Blow-up.....	36



LIST OF FIGURES

(Continued)

Figure 19.	I-80 Antelope Road Weigh Station	37
Figure 20.	SWIM Data Acquisition Cabinet Configuration	41
Figure 21.	ATC SWIM System Data Flow Diagram	41
Figure 22.	California Department of Transportation Traffic Signal Equipment Specification Input File	43
Figure 23.	Peak Hold Detector Block Diagram	45
Figure 24.	Peak Hold Detector Input Signal (Channel 1) and Output Signal (Channel 2)	45
Figure 25.	PAT Equipment Corporation, Inc., Weigh Pad Assembly	48
Figure 26.	PAT Equipment Corporation, Inc., Weigh Pad Preamplifier Interface Card Block Diagram	48
Figure 27.	Five Axle Truck Weigh Pad Output Signal (Channel 1) and Expanded Weigh Pad Output Signal (Channel 2)	49
Figure 28.	International Road Dynamics, Inc., Replaceable Dynax Axle Sensor Assembly	51
Figure 29.	International Road Dynamics, Inc., Replaceable Dynax Axle Sensor Output Signal	51
Figure 30.	Detector Systems, Inc., Model 222 Loop Detector Output Signal (Channel 1) and Weigh Pad Output Signal (Channel 2)	55
Figure 31.	Basic ATC SWIM System Configuration	58
Figure 32.	ATC SWIM Prototype System Installation	59
Figure 33.	ATC SWIM Prototype System Operational Concept	62
Figure 34.	ATC Prototype SWIM Software Sensor 0 Service Routine	62



LIST OF FIGURES

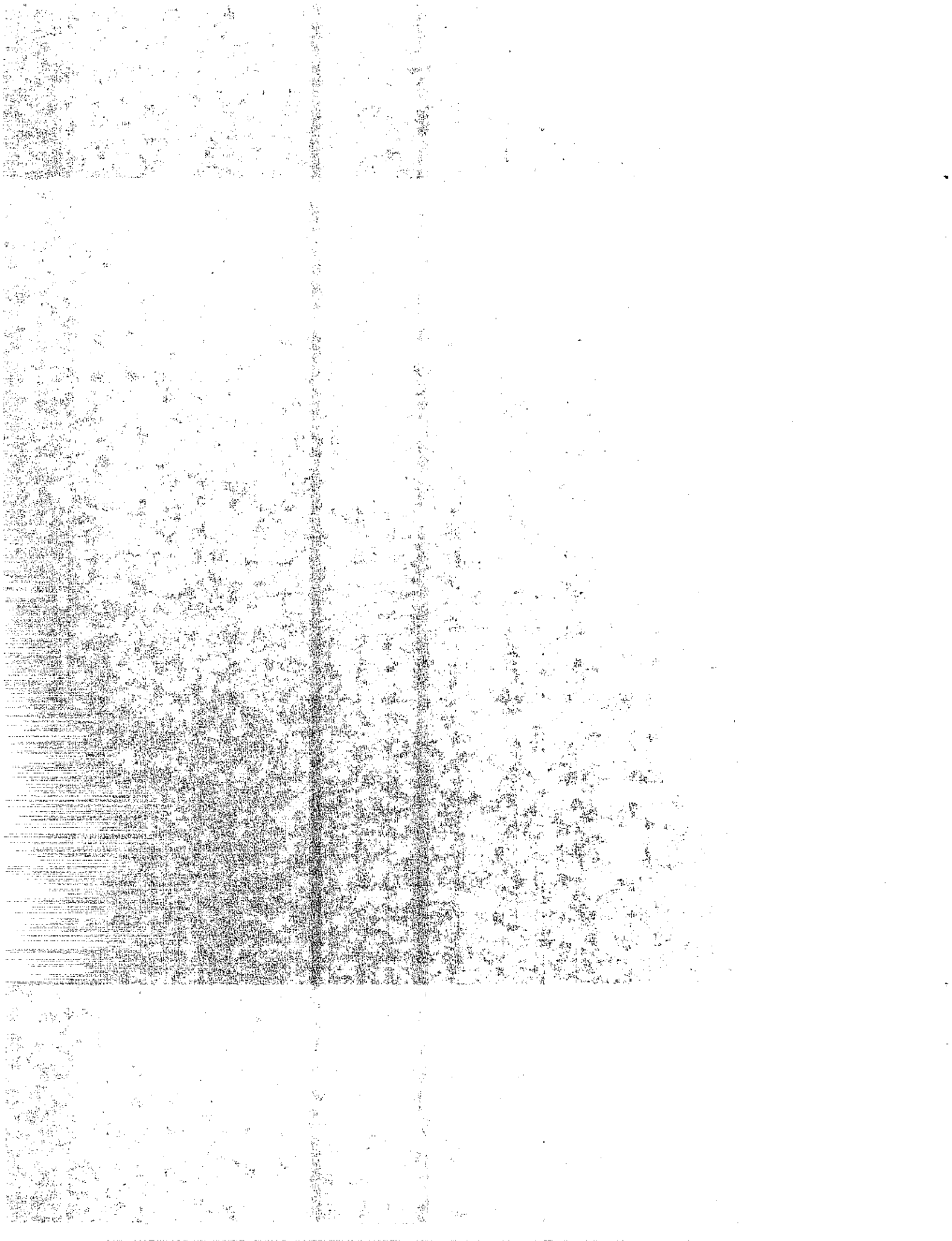
(Continued)

Figure 35.	ATC Prototype SWIM Software Sensor 1 -5 Service Routine	63
Figure 36.	ATC Prototype SWIM Software Service Sensor 6 Routine	63
Figure 37.	ATC Prototype SWIM Software Service Sensor 7 Routine	64
Figure 38.	ATC Prototype SWIM Software Service Sensor 8, 9, 10 Routine	64
Figure 39.	ATC Prototype SWIM Software Vehicle Loop Detector Service Routine	65
Figure 40.	ATC Prototype SWIM Software VDAD Routine	65
Figure 41.	ATC Prototype SWIM Software Service Sensor 1 - 5 Routine	66
Figure 42.	ATC Prototype SWIM Software Service Sensor 1 - 5 Routine Continued	66
Figure 43.	ATC Prototype SWIM Software Service Sensor 1 - 5 Routine Continued	67
Figure 44.	ATC Prototype SWIM Software Service Sensor 6 - 10 Routine	67
Figure 45.	ATC Prototype SWIM Software Service Sensor 6 - 10 Routine Continued	68
Figure 46.	ATC SWIM System Truck Record Display Screen	68
Figure 47.	Controlled Data Set - Average Percent Error versus Speed	76
Figure 48.	Controlled Data Set - Average Percent Error versus Axle Spacing, Axle Weight, and Gross Weight	77
Figure 49.	Controlled Data Set - Standard Deviation of Mean Error versus Axle Spacing, Axle Weight, and Gross Weight	78
Figure 50.	Controlled Data Set - Standard Deviation of Mean Error versus Speed	78
Figure 51.	Random Data Set - Average Percent Error versus Class	80

LIST OF FIGURES

(Continued)

Figure 52.	Random Data Set - Average Percent Error versus Axle Spacing, Axle Weight, Gross Weight, and Gross Length with Respect to Class	81
Figure 53.	Random Data Set - Standard Deviation of Mean Error versus Class	82
Figure 54.	Random Data Set - Standard Deviation of Mean Error versus Axle Spacing, Axle Weight, Gross Weight, and Over-all Length with Respect to Class	82
Figure 55.	Random Data Set - Average Percent Error versus Speed	83
Figure 56.	Random Data Set - Average Percent Error versus Axle Spacing, Axle Weight, Gross Weight, and Over-all Length with Respect to Speed	84
Figure 57.	Random Data Set - Standard Deviation of Mean Error versus Speed	85
Figure 58.	Random Data Set - Standard Deviation of Mean Error versus Axle Spacing, Axle Weight, Gross Weight, and Over-all Length with Respect to Speed	85



LIST OF TABLES

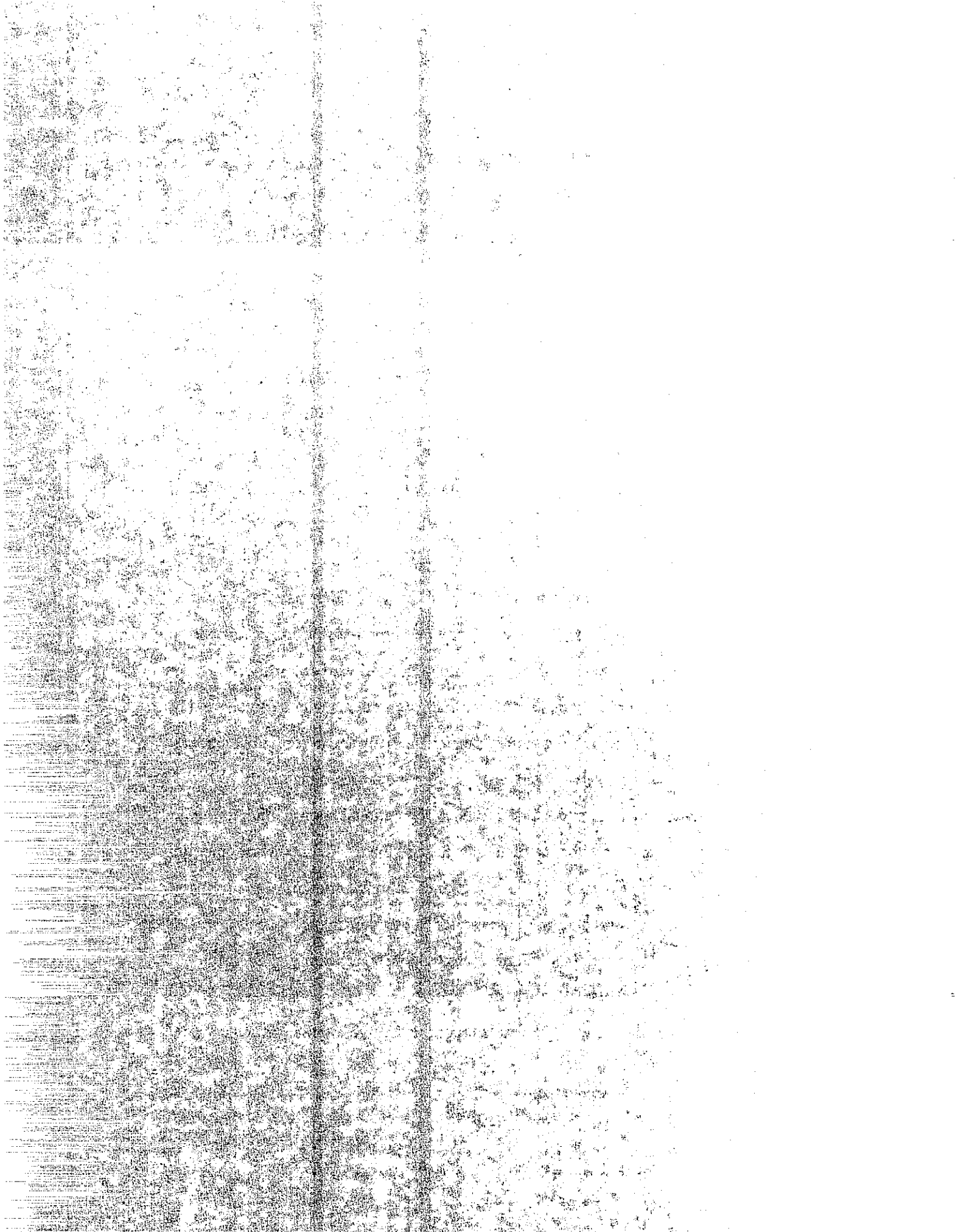
Table 1.	Antelope ATC SWIM Site Vehicle Speeds	38
Table 2.	Peak Hold Detector Board Input File Wiring	44
Table 3.	Weigh Pad Amplifier Interface Board Input File Wiring	49
Table 4.	Axle Sensor Interface Boards Input File Wiring	52
Table 5.	Loop Detector Input File Wiring	55
Table 6.	California Department of Transportation Required WIM System Accuracy's for Systems Operating with Speeds Ranging Between 3 and 40 mph.	71
Table 7.	American Society for Testing and Materials WIM System Accuracy's for Systems with Operating Speeds Between 15 and 50 mph.	71
Table 8.	FHWA Vehicle Classifications Definitions	72
Table 9.	FHWA Vehicle Weight/Axle Spacing Classifications	74
Table 10.	Controlled Data Set Average Error	76
Table 11.	Controlled Data Set Standard Deviation of Average Error	77
Table 12.	Random Data Set Average Error with Respect to Class	80
Table 13.	Random Data Set Standard Deviation of Average Error with Respect to Class	81
Table 14.	Random Data Set Average Error with Respect to Speed	83
Table 15.	Random Data Set Standard Deviation of Average Error with Respect to Speed	84

EXECUTIVE SUMMARY

This report presents the development and evaluation of an Advanced Transportation Control Computer (ATC) specifically designed for use in Intelligent Vehicle Highway Systems (IVHS).

The California Department of Transportation's current control computer is designed for basic signalized intersection operation and is not appropriate for complex IVHS strategies. The ATC however, is designed for advanced applications requiring fast processing, extended memory, state-of-the art communication capabilities, and overall system flexibility. This report describes the ATC hardware and software, its capabilities and limitations, and its performance as demonstrated in a slow speed weigh-in-motion (SWIM) field test environment. The ATC will serve as a basis for the development of a standard IVHS controller.

Recently, a variety of commercial IVHS applicable controllers have become available. Unfortunately, these machines only provide a stopgap solution, as they are typically proprietary and are designed with incompatible hardware and software systems. The ATC however, is designed with a set of standard components which can be used by a variety of transportation agencies, providing transparent operation across jurisdictional boundaries. Based on the success of standards in other industries, a standard transportation controller will promote functional and operational coordination throughout the transportation system. Potential savings in the Department's maintenance operations over the next ten to fifteen years alone could be in the millions of dollars. An inventory of standard controller units and plug-in modules carried on maintenance vehicles, will promote efficient troubleshooting and minimize down-time significantly. Savings in software development could also be substantial. High-level language and operating system software will permit easy migration from one system to another, offer superior



development tools, reduce programming complexity, time, and expense, and promote reusable software.

The ATC prototype hardware and software is based on an Institute of Electrical and Electronic Engineers standard 3U VME bus, Motorola 68020 CPU, and Microware OS-9 C programming language and real time operating system. The cost of the control hardware and software needed for the SWIM demonstration totaled \$6,280 (1990 quantity one price with special purpose SWIM cards). Because computer pricing continues to decline steadily, future pricing is likely to be less than \$2,000 per unit. In proprietary systems, such as SWIM applications, where pricing of \$30,000 per unit is typical, substantial savings can be realized.

The results of this study are grouped into two categories: ATC performance and SWIM system performance. ATC performance was based on qualitative criteria (i.e. ease of software development, number and type of hardware problems encountered, user feedback, vendor support, etc.). SWIM performance was based on a quantitative evaluation (i.e. statistical compliance with the Department's WIM accuracy criteria).

Overall, the ATC hardware and software performed quite well. Although somewhat complex, the VME based ATC provided a clean sturdy design available through numerous vendors. The controller functioned properly in extreme environmental conditions and the modular configuration proved useful in troubleshooting and maintenance. Some difficulties were encountered in software development, as project engineers, new to real-time programming, experienced a rather steep learning curve.

SWIM system performance was mixed. Although axle spacing measurements fell well within the Department's accuracy limits, weight measurements were slightly greater than allowed by the Department. Test results indicated SWIM system performance to be a function of speed, as both controlled and random data sets consistently reflected greater inaccuracies for specific speed groups. This observation, also noted in other WIM



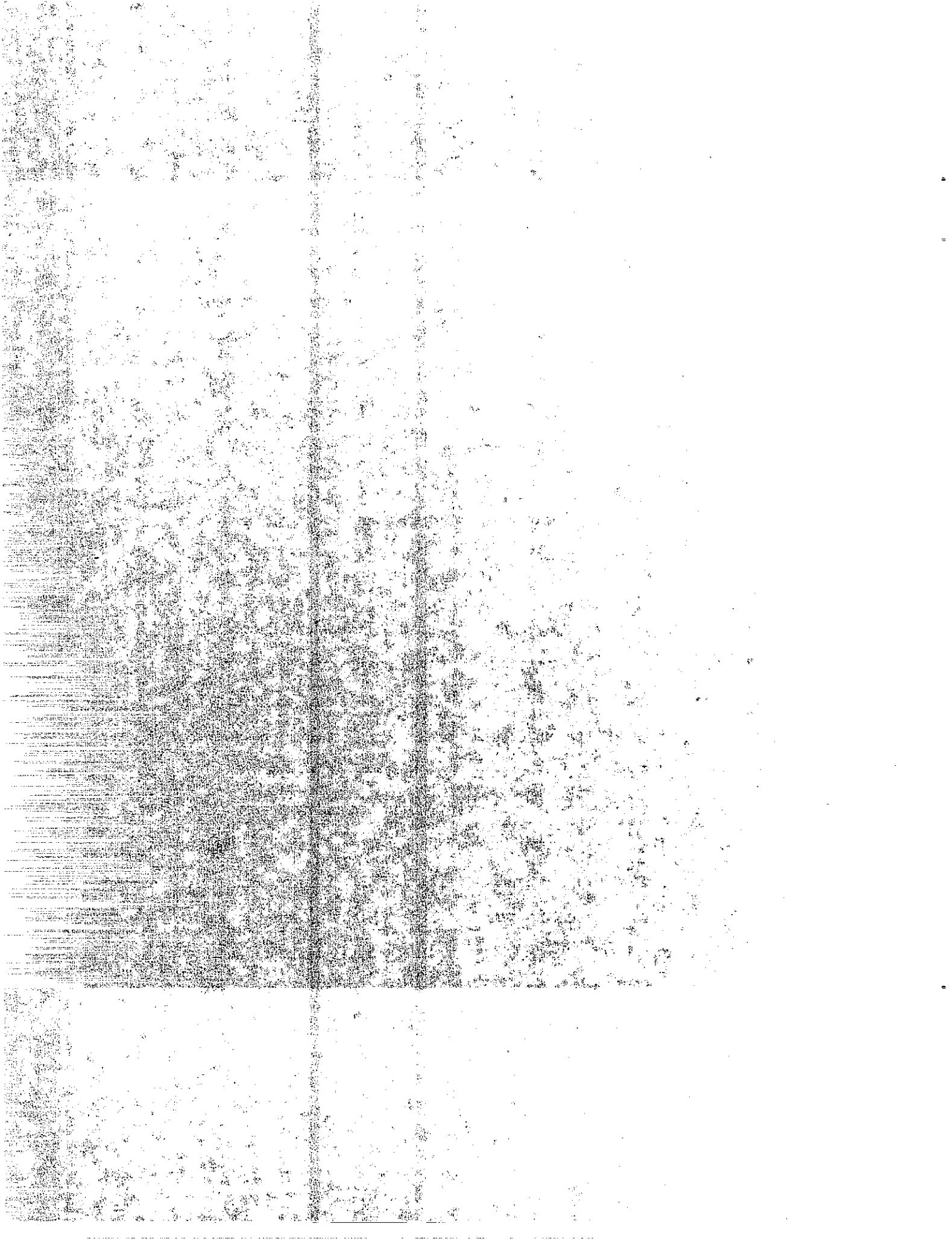
studies, is likely due to the interaction of the vehicle suspension and pavement irregularities, causing bouncing and unpredictable axle loading.

Performance results lead to the following general recommendations:

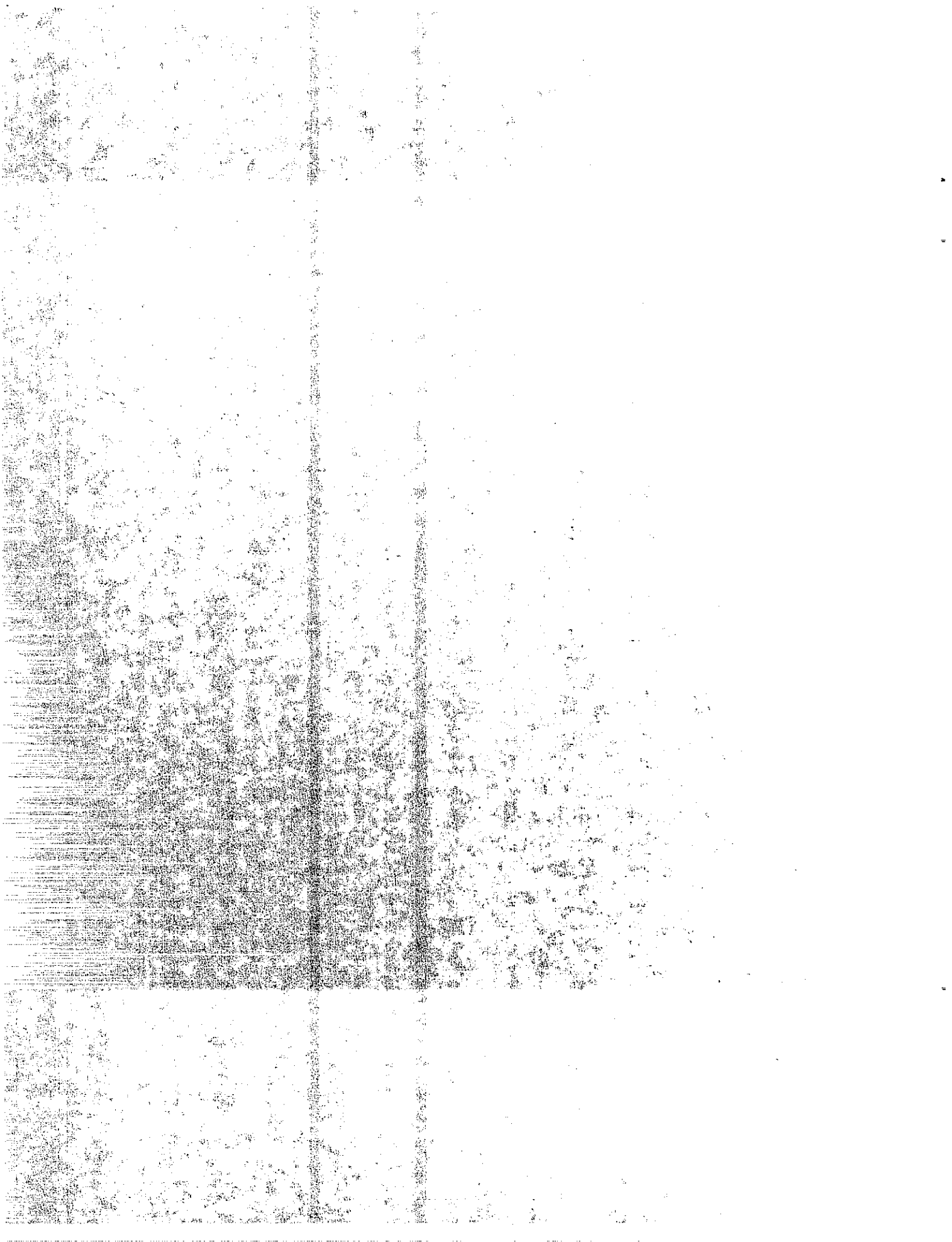
ATC Compatible Hardware and Software: The standard controller should support a 3U VME backplane, Motorola based CPU, C programming language, and OS-9TM real time operating system. The CPU module should provide a Motorola 68020 or greater microprocessor, with consideration given to the 683XX family and CPU32 instruction set. A minimum of two serial ports should be provided. Use of low power components should be maximized. Software standards should be maximized.

SWIM: The SWIM system should be designed such that vehicles are channeled into the SWIM detection zone and are not permitted to leave the active detection area until all data has been collected. The roadway geometry should be designed such that vehicles have enough room to approach steady speeds, minimizing accelerations and decelerations.

Several activities are currently underway to aid in ATC implementation. A Controller Development Team has been formed by the Department to develop formal specifications for an Advanced Transportation Management System Controller which will incorporate many of the concepts discussed herein. The Department has formed a Communication Committee to address advanced controller communication issues: controller-to-controller, controller-to-master, master-to-master, and master-to-Traffic Operations Center. The Department will make available a report on object-oriented application development software, designed to run on the ATC platform, entitled "A Model for Roadway Traffic Control Software". A Federal Highway Administration IVHS Field Operational Test in Orange County specifies the ATC in the implementation of an Integrated Freeway Ramp Metering and Arterial Adaptive Signal Control system. Signalized intersection operations have been implemented on ATC hardware in the City of Los Angeles.



As indicated by the level of activity in this area, the need for a standard IVHS applicable controller is tremendous. The lack of such a standard results in expensive incompatible proprietary systems, increased maintenance and operations costs, a proliferation of throw-away software, and poor quality control. The ATC prototype, providing a basis for the development of standard specifications, test procedures, and software, is designed to eliminate many of these problems.



1. INTRODUCTION

1.1 Problem

The current Caltrans standard traffic control computer was designed for simple signalized intersection control and is not appropriate for complex transportation strategies required in Intelligent Vehicle Highway Systems (IVHS). Advanced applications demand more speed, memory, communication channels, and system flexibility. Until recently, commercial controllers specifically designed for advanced transportation applications were not available. As a result, a variety of devices emerged to fill this void. These machines only provide a stopgap solution however, as hardware and software compatibility does not exist. An advanced transportation controller *standard* is needed to reduce hardware and software development costs, promote compatibility in maintenance and operations, and increase overall system reliability and efficiency.

1.2 Objective

The goal of this project, to design and implement a real-time control computer capable of supporting state-of-the-art hardware and software, was achieved through the development of a prototype computer and the demonstration of prototype capabilities in a slow speed weigh-in-motion (SWIM) field test environment. The computer served as a basis for the development of standard specifications, test procedures, and software.

1.3 Background

Caltrans operates thousands of Model 170 microprocessor-based traffic control units. Virtually all State operated signalized intersections and ramp metering systems rely on the 170 controller designed by Caltrans in 1976. During the seventeen years since its inception however, major advancements in hardware and software technology have taken place which can be used to implement new IVHS applications and can significantly increase system efficiency and effectiveness.

Several IVHS related projects exist in California today and many are planned for the future. Systems include High Occupancy Vehicle (HOV) reversible lanes, High speed Weigh-In-Motion (WIM) with Automatic Vehicle Identification (AVI), Integrated Ramp Metering and Arterial Adaptive control, and general Traffic Management Systems (TMS) encompassing these and other advanced concepts. Although specific requirements differ for these systems, all rely on computers excelling in speed, memory, and communication capabilities for control of electronic devices such as: optical and inductive-loop vehicle detectors, closed circuit televisions, changeable message signs, weigh pads, axle sensors, highway advisory radios, RF (radio frequency) AVI readers, and communication equipment. Unfortunately, the lack of appropriate control equipment standards has forced designers to invest in custom and proprietary hardware and software. This results in expensive incompatible systems, increased maintenance and operation costs, a proliferation of throw-away software, and virtually no quality control. The ATC prototype controller, providing the basis for an IVHS applicable standard, is designed to eliminate many of these problems.

Slow Speed Weigh-in-motion (SWIM), selected for the ATC prototype test, provides a good example of problems associated with incompatible proprietary systems. The need for SWIM truck screening in California weigh stations is urgent, as over 250 million commercial trucks travel the State's highways annually. It has become necessary in many cases to bypass large numbers of trucks to prevent hazardous back-up conditions on main-line freeway lanes. Without adequate weight enforcement, increased highway maintenance and reconstruction will be required to ensure safe roadway conditions.

This urgency has prompted the installation of a variety of proprietary SWIM systems throughout the State. Proprietary controllers often cost \$30,000 per unit, or more, and restrict in-house maintenance, requiring contracts in excess of \$100,000 annually. When failures do occur, downtime may be lengthy as manufacturers are often foreign based. Proprietary systems also restrict the

Department from modifying software to meet changing needs and provide little or no uniformity from one system to the next.

This project provides a solution to proprietary and incompatible SWIM systems while demonstrating the capabilities of the proposed ATC platform. A fully functional SWIM system was developed based on traditional weigh pad and axle sensor hardware and a prototype ATC SWIM controller. SWIM software was developed by the Department on the ATC to provide weight, axle spacing, classification, and violation information. Technical details related to general purpose ATC hardware and software are found in Section 4.1 and SWIM specific hardware, software, and system accuracy information is found in Section 4.2. Detailed data sheets and source code are provided for all hardware and software in the appendices. Conclusions, recommendations, and implementations are discussed in the following two sections.

2. CONCLUSION

2.1 Prototype Performance

The results of this study can be grouped into two categories: i) ATC performance and ii) Slow Speed Weigh-In-Motion (SWIM) performance. ATC performance was based on qualitative criteria (i.e. ease of software development, number and type of hardware problems encountered, user feedback, vendor support, etc.). SWIM performance was based on a quantitative evaluation (i.e. statistical compliance with the Department's WIM accuracy criteria).

2.1.1 ATC Hardware and Software

ATC prototype hardware and software, as described in Section 4.1 and shown in Figure 4, page 17, based on the 3U VME bus, Motorola 68020 CPU, C programming language, and OS-9TM* real time operating system performed well. The field controller hardware, at a cost of \$6,100 (1990 quantity one price with special purpose SWIM cards), included a 68020 CPU card with two serial ports, an analog to digital converter card, two digital input cards, a power supply, a 15 slot backplane, and a 19 inch rack mount chassis. Controller software, at a cost of \$180 (quantity one), included industrial OS-9TM real-time operating system. Because controller pricing has declined substantially, mirroring price reductions common in the consumer electronics industry, future pricing of less than \$2,000 per unit is anticipated. In applications such as WIM, where \$30,000 per unit is typical, substantial savings can be realized.

HARDWARE

Backplane

Although somewhat complex with 96 lines, the VME backplane provided a clean sturdy design available through numerous manufacturers. Because only four of the fifteen available slots were

*OS-9 is a trademark of Microware Systems Corporation

used in the I/O intensive SWIM application, future general purpose controller specifications could safely require fewer slots, saving space and reducing costs even further. Another backplane observation involved termination. Passive termination, used to prevent ringing and described in Section 4.1.1, page 12, actually increases power consumption, at a continuous six Watt loss. Whereas active termination, peaking at six Watts, only experiences a continuous two to three Watt loss. Active termination should be considered in future applications.

CPU Module

The compact 3U VME boards, 3.937 inch by 6.299 inch, were easy to handle and required minimal cabinet space. The number of 3U manufacturers however are few compared to 6U manufacturers and many high-end boards, such as video and communication processors, are not readily available in the 3U form factor. Despite 6U availability, the ATC 3U CMOS cards, unlike 6U cards, provide a compact low power solution ideal for field applications (CPU card < 5 Watts, support cards < 1 Watt each). Since the purchase of the ATC 68020 CPU Card, newer 3U 683XX cards, based on Motorola's CPU32 core, have become quite popular. CPU32 is a 68000/10 compatible CPU that executes instructions with 68020 performance. CPU32 cards typically provide a number of functions and offer very low power CMOS operation. CPU32 card designs should be considered in future applications.

Power Supply

The modular power supply concept proved useful in troubleshooting and maintenance. A suspected faulty supply was replaced with an identical plug-in module, effectively eliminating the problem. The failure was determined to be related to line stability. Future designs should clearly specify acceptable voltage swings to ensure DC levels remain within the tolerance required by most ICs.

Environmental

Although the prototype did not specify extended temperature ranges or other provisions for extended environmental conditions, the controller appeared unaffected by either 100+ °F heat or extreme moisture. Initial tests were performed in late summer during average high Sacramento Valley temperatures of 95 - 105 °F. It is safe to assume internal cabinet temperatures were much greater. No problems were observed. Testing continued into the fall rain season. Due to poor weather stripping and ventilation in the 10+ year old equipment cabinet, standing water was found throughout the controller assembly. Again no problems were observed.

Software

Software development was more time consuming than expected. Project engineers, new to real-time software in general, experienced a rather steep learning curve. Some problems were also encountered with the operating system development environment. Documentation was difficult to use (i.e. limited indexing, poorly organized, few examples, etc.) and development tools were limited (i.e. little on-line help, unpredictable cross development package, no windowing capabilities, etc.). Excellent software support services were available however through the operating system vendor, the hardware manufacturer, and other Caltrans personnel experienced in real-time programming. In-house software experience proved invaluable and should be a key factor in software selection. The ability to develop both in resident and non-resident environments also helped the development process by providing a convenient degree of flexibility. C code could be developed in a "favorite" environment and later ported to the VME platform and/or developed directly on the ATC machine. Once in the field, few software changes, other than calibrations, were required. This real-time software programming experience supports the need for user friendly object-oriented application development software as described in Section 4.1.2.3.

2.1.2 SWIM System

Prototype SWIM axle spacing accuracies fell well within the Department's accuracy limits. Section 4.2.4, Figures 47 and 48, pages 76 and 77 respectively, show a controlled data set (multiple passes of same vehicle), with an average axle spacing error of 0.48%, equating to 0.07 feet, where the Department's accuracy standard is 0.5 feet. However, weight accuracies did not meet the Department's criteria. Single axle weight errors for the same data set resulted in an average of 3.89% versus the Department's accuracy standard of 2.00%. Gross weight errors measured 3.90% versus the Department's accuracy standard of 2.00%. Similar results were found for a random set of trucks, as described in Section 4.2.2 and shown in Figures 51 through 58, pages 80 and 85 respectively. Axle spacing error measured 1.18%, equating to 0.47 feet, compared to the Department's accuracy limit of 0.50 feet. Over all spacing error measured 0.76%, equating to 0.57 feet, where the Department's accuracy standard is 1.00 foot. Average random vehicle weight errors were 4.07% for single axles versus the Department's accuracy requirement of 2.00%. Gross weight average error measured 3.39% versus the Department's accuracy standard of 2.00%. Test results indicated SWIM system performance to be a function of speed, as both controlled and random data sets consistently reflected greater inaccuracies between 10 and 20 mph. This observation, noted in other WIM studies, is likely due to the interaction of vehicle suspensions and pavement irregularities which cause bouncing and unpredictable axle loading. The Antelope site exhibits a 3% cross slope and a pavement profile index (measure of smoothness of pavement) of 25.5 inches/mile, both exceed recommended conditions. These factors were believed to contribute to the weight inaccuracies.

2.2 Recommendations

2.2.1 ATC Hardware and Software

Based on the success of standards in other hardware and software industries, an ATC standard will likely promote functional and operational coordination statewide. Potential savings in maintenance

operations could be in the millions of dollars over the next ten to fifteen years. An inventory of standard controller units and plug-in modules carried on maintenance vehicles, would promote efficient troubleshooting and minimized down-time significantly. Savings in software development could also be substantial. High-level language and operating system software will permit easy migration from one system to another, offer superior development tools, reducing programming complexity, time, and expense, and promote extensible reusable software. The following recommendations support the standardization of an Advanced Transportation Controller.

- * The controller should support a standard 3U VME backplane, Motorola based CPU, C programming language, and OS-9 TM real time operating system as detailed in Section 4.1.
- * The CPU module should provide a Motorola 68020 or greater microprocessor, with consideration given to the 683XX family and CPU32 instruction set. A minimum of two serial ports should be provided. Use of low power components should be maximized.
- * The backplane, for general purpose applications, should require five slots, with provisions for more as required by high-end applications. Active termination should be utilized.
- * The power supply should be modular and limit voltage swings to ensure DC levels remain within tolerance required by most ICs.
- * A 6U VME system is recommended for laboratory experimentation, software development, and applications where 3U boards are not readily available.
- * Software standards should be maximized. An object-oriented programming environment should be employed for application development whenever possible.

- * Standard hardware and software test methods should be developed and incorporated into a quality assurance program. Standard diagnostic software should be provided for laboratory and field testing.
- * A training program should be developed for maintenance and operations personnel.

2.2.2 SWIM System

The ATC SWIM system produced mixed results. Although axle spacing accuracies demonstrated the processing capability of the ATC controller, weight inaccuracies, lead to the following SWIM site recommendations.

- * Design roadway geometry of new SWIM systems such that vehicles are channeled into the SWIM detection zone and are not permitted to leave the active detection area until all data has been collected.
- * Design roadway geometry of new SWIM systems such that vehicles do not form a queue and have ample room to reach a steady speed, minimizing acceleration and deceleration.
- * Ensure SWIM system roadway designs minimize pavement irregularities and meet the Department's maximum profile index.
- * Ensure SWIM system roadway designs comply fully with weigh pad manufacturers installation criteria.

3. IMPLEMENTATION

The results of this study will be distributed to the Department's Division of Traffic Operations, Transportation Planning, New Technology, Materials and Research, and Highway Planning and Research. The results of this study will also be distributed to the California Highway Patrol (Commercial and Technical Services Division), and interested states, regions, and local agencies on request.

A Controller Development Team (CDT) has been formed to develop formal specifications for an Advanced Transportation Management System (ATMS) controller standard incorporating many of the concepts discussed herein. The CDT consists of representatives from Caltrans Districts 04, 07, 11, and 12, Headquarters Traffic Operations Electrical Systems Office, New Technology Materials and Research Electrical and Electronics Engineering Office, the City of Los Angeles, Transportation Department, and JHK and Associates as consultants. A draft specification is due for release in September 1993, with a final procurement specification available in the first quarter of 1994.

The Department has formed a Communication Committee to address advanced controller communication issues: controller-to-controller, controller-to-master, master-to-master, and master-to-Traffic Operations Center (TOC). An investigation of existing communication protocols appropriate for advanced controller use was conducted and a proposed protocol, based on the study findings, has been released for review and comment.. The Department also remains in contact with the National Electrical Manufacturers Association (NEMA) to promote compatible protocols.

The Department will make available a report on object-oriented application development software, designed to run on the ATC platform, entitled "A Model for Roadway Traffic Control Software". This software was developed for Caltrans by UC Irvine, Carnegie Mellon University, and

Louisiana State University. The report will describe the conceptual organization of the object-oriented package including a synchronous data flow model of the Traffic Control Blocks (TCBLKS), implementation of the TCBLKS, scheduling of tasks in the TCBLK model, examples, and an evaluation.

A graphical user interface (GUI) has been developed by the Department for the object-oriented software described above, and in Section 4.1.2, page 29, to form a complete user friendly development package specifically targeting the ATC platform. A Ramp Metering application implemented along U.S. 50 in Sacramento demonstrated the strength of this software package and the ATC hardware. A High Speed Weigh-in-Motion application, to be implemented by the Department along I-5 in Lodi California, will also take advantage of this software development tool and the ATC platform. Widespread use of this software is anticipated.

A Federal Highway Administration (FHWA) Intelligent Vehicle Highway System (IVHS) Field Operational Test in Orange County specifies the ATC in the implementation of an Integrated Freeway Ramp Metering and Arterial Adaptive Signal Control system. The test will integrate an existing centrally controlled freeway ramp metering system with arterial signals consisting of existing signal controllers (both NEMA and 170) and the ATC running an Optimized Adaptive Control (OPAC) algorithm.

Signalized intersection operations have been implemented on ATC hardware. The ATC-1, developed by the City of Los Angeles, is a prototype ATC designed to operate in existing traffic signal equipment cabinets as a direct replacement for the Model 170 controller. In addition to the basic ATC hardware platform, the ATC-1 provides a four line by forty character illuminated liquid crystal display and user friendly interface software. The eight phase control algorithm was written in C.

4. TECHNICAL DISCUSSION

4.1 ATC Prototype

Although a number of hardware and software configurations are possible in advanced control applications, the ATC prototype is based on a modular design divided into five functional areas: i) standard data bus, ii) microprocessor module, iii) input/output module, iv) support hardware, and v) high-level programming language and real-time operating system software. The prototype, designed for a Slow Speed Weigh-in Motion (SWIM) application, includes a 3U VME bus, Motorola 68020 microprocessor, RS-232 serial communication interface, TTL level digital I/O, 12 bit analog to digital converter (A/D), C mid-level programming language, and OS-9TM real-time operating system. The following sections detail the prototype configuration.

4.1.1 Hardware

Criteria for selecting computer control hardware depends on a variety of factors including: computational ability, memory capacity, number of input/output and communication channels, adherence to standards, flexibility, reliability, environmental adequacy, equipment availability, support service, and cost. These and other selection criteria were considered in the ATC prototype design. Appendix A provides detailed data sheets for all hardware modules. Prototype specifications are described below.

4.1.1.1 Backplane

The ATC prototype specifies a 3U VME bus. VME, a descendant of VERSAbus-E developed by Motorola in the late 1970s, is an ANSI/IEEE standard #1014-1987(1). The bus, also referred to as a backplane, interfaces data collection, storage, processing, and peripheral control devices via four bus lines: i) data transfer, ii) interrupt, iii) arbitration, and iv) utility. Figure 1 depicts these buses and their interaction with controller modules(1).

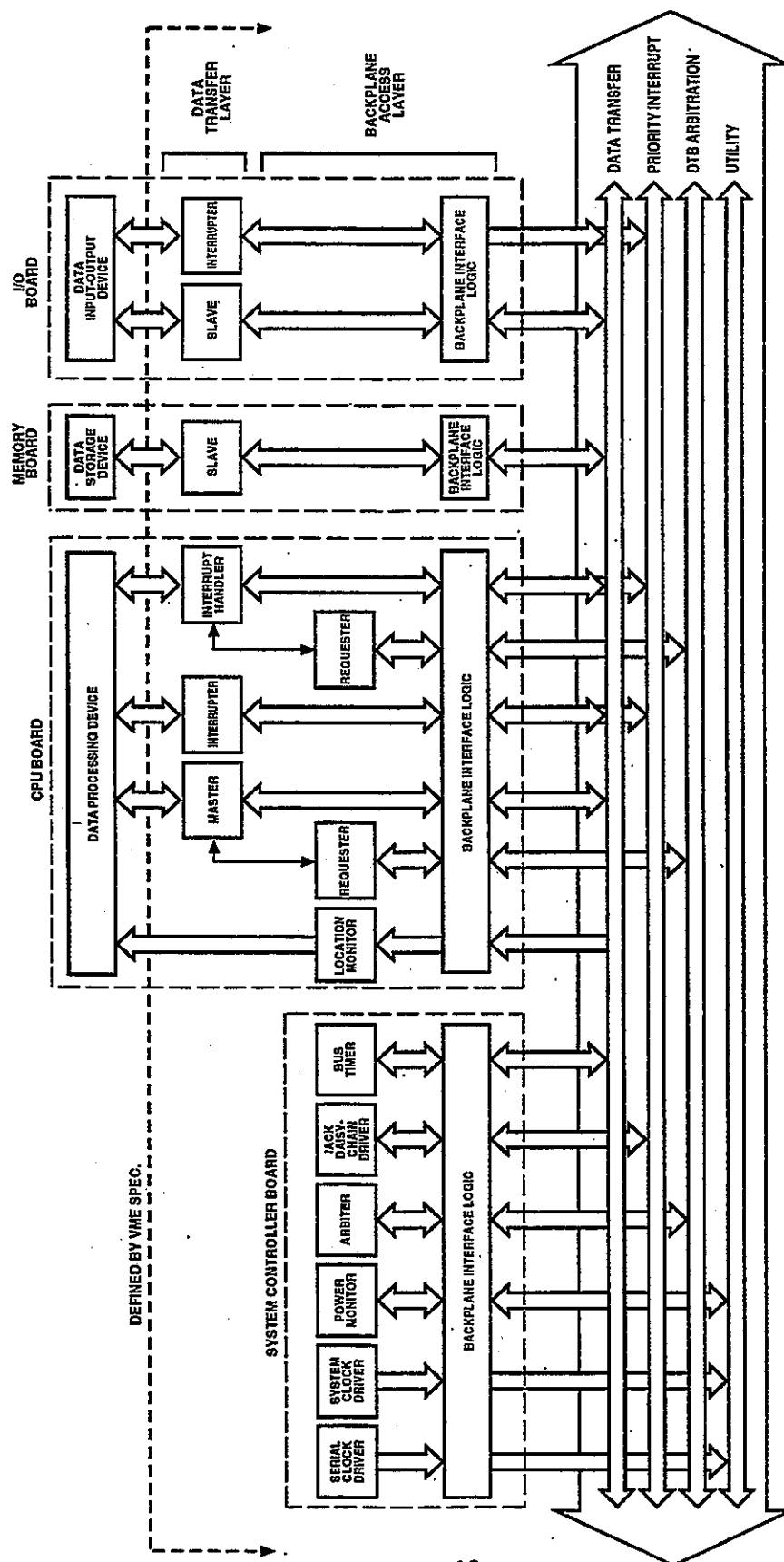


Fig. 1. FUNCTIONAL MODULES AND BUSES DEFINED BY VME BUS IEEE STANDARD

In addition to data lines D00-D31 (3U D00-D15) and address lines A01-A31 (3U A01-A23), the data transfer bus (DTB) uses six control lines similar to Motorola's 68000 CPU signals, AS* (Address Strobe - indicates an address is stable and available for use), DS0* and DS1* (Data Strobe Zero and Data Strobe One - selects locations for data transfer and indicates availability of valid data on the data bus), BERR* (Bus Error - indicates unsuccessful data transfers), DTACK* (Data Transfer Acknowledge - indicates successful read or write of data), WRITE* (Read/Write - indicates data transfer direction: low status indicates master to slave transfer, high status indicates slave to master transfer). DTB also uses address modifier signals AM0-AM5 to indicate the type of access (user, supervisory, code, data, or stack) and a long word signal LWORD* for 32-bit transfers.

The interrupt bus has seven interrupt request lines IRQ1*-IRQ7*, each corresponding to an interrupt priority (IRQ7* being the highest) with up to 256 interrupt vectors per level. Daisy chain IACKIN* and IACKOUT* signals ensure that if more than one board issues an interrupt with the same priority level at the same time, only one will respond to the acknowledge. An interrupt acknowledge IACK* signal indicates a successful interrupt.

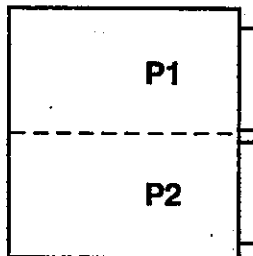
The arbitration bus prevents use of the bus by two master processors at the same time and optimizes use of the bus in multiprocessing systems. Four levels of arbitration are supported, each providing Bus Request signals (BR0* - BR3*) and Bus Grant daisy chain signals (BG0IN* - BG3IN* and BG0OUT* - BG3OUT*). Bus grants for multiple masters are based on three different scheduling algorithms.

* indicates active low signals.

The utility bus handles System Clock (SYSCLK), Serial Clock (SERCLK), Serial Data (SERDAT*), System Reset (SYSRESET*), System Failure (SYSFAIL*), AC Failure (ACFAIL*), and power supply. SYSCLK and SYSFAIL* can be used by any board in the system. The SYSCLK driver is positioned on the system controller in slot number 1. A power monitor detects power failures, abort/reset button activity, and initiates orderly system shutdown.

All signals are transferred from the VME bus to controller modules via 96 pin pin-in-socket connectors. The ATC field controller specifies a single connector "single height" 3U backplane (P1) at 3.937 inches which supports 24 bit addressing and 16 bit data transfer. A double connector "double height" 6U backplane (P1 and P2) at 9.187 inches extends addressing and data transfer to a full 32 bits. A double height chassis with a P1 backplane is specified in the development system. This hardware configuration is discussed in more detail in the Support Hardware section, page 25. Both 3U and 6U height boards measure 6.299 inches in depth. Both field and development systems specify a 15 connector (slot) backplane. Figures 2 and 3 illustrate these configurations(1). Figure 4 depicts the prototype 6U ATC development machine and the prototype 3U ATC field controller. Figure 5 depicts 3U VME bus connectors and chassis. For complete VME bus information refer to IEEE Standard 1014 (1).

P1 / J1 AND P2 / J2 CONNECTORS



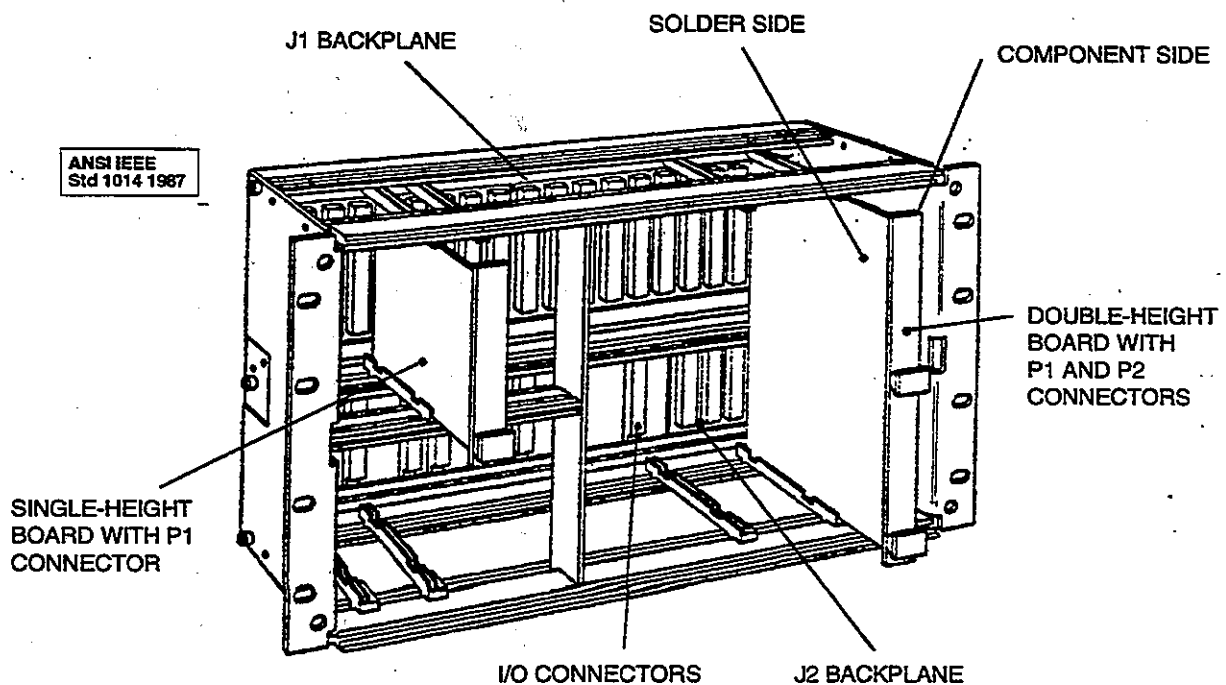
● P1 / J1 CONNECTOR

- SUPPORT 24 BIT ADDRESS & 16 BIT DATA CAPABILITY
- ALL CONTROL AND OTHER FUNCTIONS

● P2 / J2 CONNECTOR

- EXTENDS TO FULL 32 BIT ADDRESS & 32 BIT DATA CAPABILITY.
- 64 USER I/O LINES

Fig. 2. P1 AND P2 VME BUS CONNECTORS



Rule 7.2 Double-height subracks shall have either (1), (2), or (3).

- (1) AJ1 backplane mounted in the upper portion of the subrack.
- (2) AJ1 and a J2 backplane, with the J1 backplane mounted in the upper portion and the J2 backplane mounted in the lower portion.
- (3) A double-height J1 / J2 backplane that provides both J1 and J2 connectors.

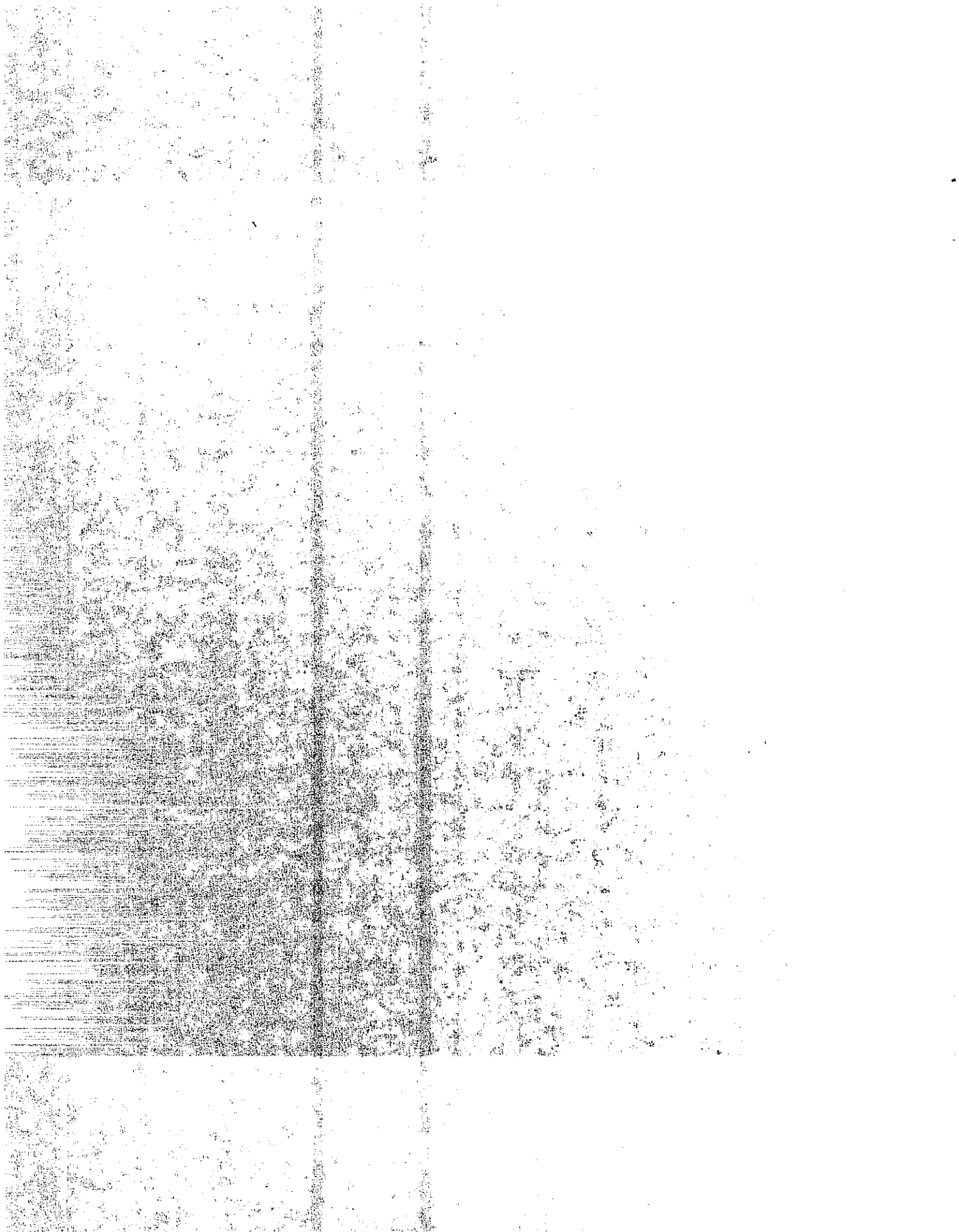
Fig. 3. VME BUS SUBRACK WITH MIXED BOARD SIZES

4.1.1.2 Microprocessor

The prototype CPU module is based on the Motorola 68020 16.67MHz CPU. The 68020 is a full 32 bit microprocessor providing 4 gigabytes of address space, an enhanced instruction set, a floating point coprocessor interface (FPCP), and a 32 bit ALU (arithmetic logic unit) for reduced processing time. The availability of 256 bytes of on-chip cache also produce significant increases in program execution speeds. The 68020 includes dynamic bus sizing with 8, 16, or 32 bit capabilities supporting dynamic memory and access to a variety of peripherals.

The prototype CPU module includes 8 MBytes DRAM, 1 KByte ROM, FPCP option, real-time battery backed clock with 12/24 hour time keeping, day-of-week counter, programmable alarm and interrupts, and two RS-232 serial ports (configurable to RS-422 and glass and plastic fiber links). The VME bus interface includes master capabilities with A24/A16 and D16/D8. Bus arbitration is configured for single level BR3* with 30 ns daisy chain logic. Power requirements are reduced to less than 5 watts with CMOS components and standard operating temperature ratings range from 0° to 70°C (with options for extended -40° to +85°C and mil spec -55° to +125°C). The front panel provides a LED Halt indicator, reset and abort push buttons, and two 15-pin sub-D sockets configured for the RS232 option. Figure 6 provides a functional description of the ATC prototype CPU module(9). Figure 7 shows the module, a PEP Modular Computer VM20 VME bus Master CPU Card.

Since the purchase of the ATC CPU module, newer 3U 683XX CPU cards, based on Motorola's CPU32 core, have come to market. The CPU32 is a 68000/10 compatible CPU that executes instructions in half the number of cycles of the 68000, resulting in 68020 performance. CPU32 cards typically provide a number of functions including DMA, counters, and timers and offer extremely low power CMOS operation at typically less than two Watts with standby conditions at typically less than 1.5 Watts.



4.1.1.3 Input/Output and Communications

As indicated in the previous section, the prototype CPU module provides several options for communications interface (RS-232, RS-422, and glass and plastic fiber). Because the prototype SWIM application is limited to an isolated location requiring no network capabilities, additional communication cards were not necessary. The simple RS-232 physical link sufficed for dumb terminal and printer interface.

Two types of I/O modules are included in the ATC prototype, an analog to digital converter (A/D) card and digital input cards. The digital input cards are daughter-boards to the PEP Modular Computer VMOD Mother-board, a "Flexible Industrial I/O Interface Module". The VMOD card is a single height board capable of accepting two industrial I/O daughter-boards or "piggybacks". The mother-board provides a VME bus slave interface A24:D16/D8 or A16:D16/D8, separate multi-level interrupt request lines for each piggyback, low power CMOS circuitry (less than 1 watt), and extended temperature range options (standard 0° to 70°C, extended -40° to +85°C and mil spec -55° to +125°C), with the standard range specified for the prototype test. Each digital I/O piggyback provides sixteen TTL-level opto-isolated inputs and four programmable edge detection/handshake lines (positive or negative going). The digital I/O mother-board/piggyback module is used in axle sensor data acquisition. The role of the module is described in the Operational Test Section 4.2.4, page 57. Figure 8 provides a functional description of the ATC prototype digital I/O mother-board and Figure 9 describes the ATC prototype digital input piggybacks(9). Figure 10 shows the mother-board and piggyback module (9).

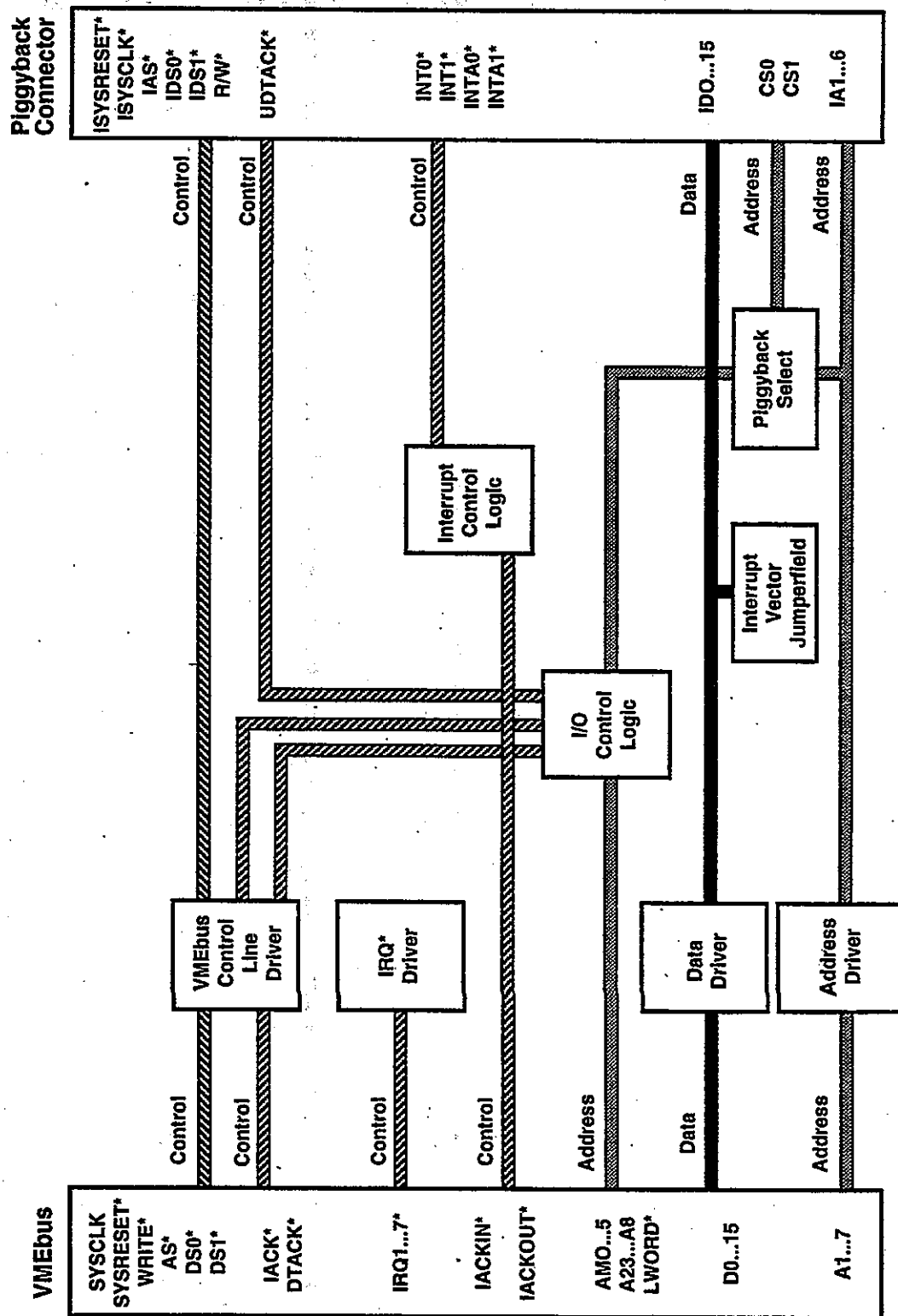


Fig. 8. FUNCTIONAL BLOCK DIAGRAM OF PROTYPE ATC I/O MOTHERBOARD MODULE

The analog to digital converter module is PEP Modular Computer's 3U VDAD card, a combined D/A, A/D, and digital I/O module; the ATC prototype does not require the D/A option. The A/D feature provides 16 single-ended or 8 differential analog inputs (differential mode is required in weigh pad data acquisition), with 12 bit resolution per channel, a 25 usec conversion time, and a programmable output voltage of 0V to 10V, -5V to +5V, or -10V to +10V (0V to 10V was specified in the SWIM application). An additional eight digital I/O lines, 3 handshake lines, and 68230 8 MHz digital timer are provided which, in conjunction with custom interface circuitry, produce required interrupt signals. A description of the interrupt mechanism is found in the Operational Test Section 4.2.4 page 57. As with other modules, the A/D card is based on low power CMOS components and consumes little power, less than 3 watts, and supports extended temperature ranges (standard 0° to 70°C, extended -40° to +85°C and mil spec -55° to +125°C), with the standard range specified for prototype testing. Figure 11 outlines the functional structure of the A/D module(9).

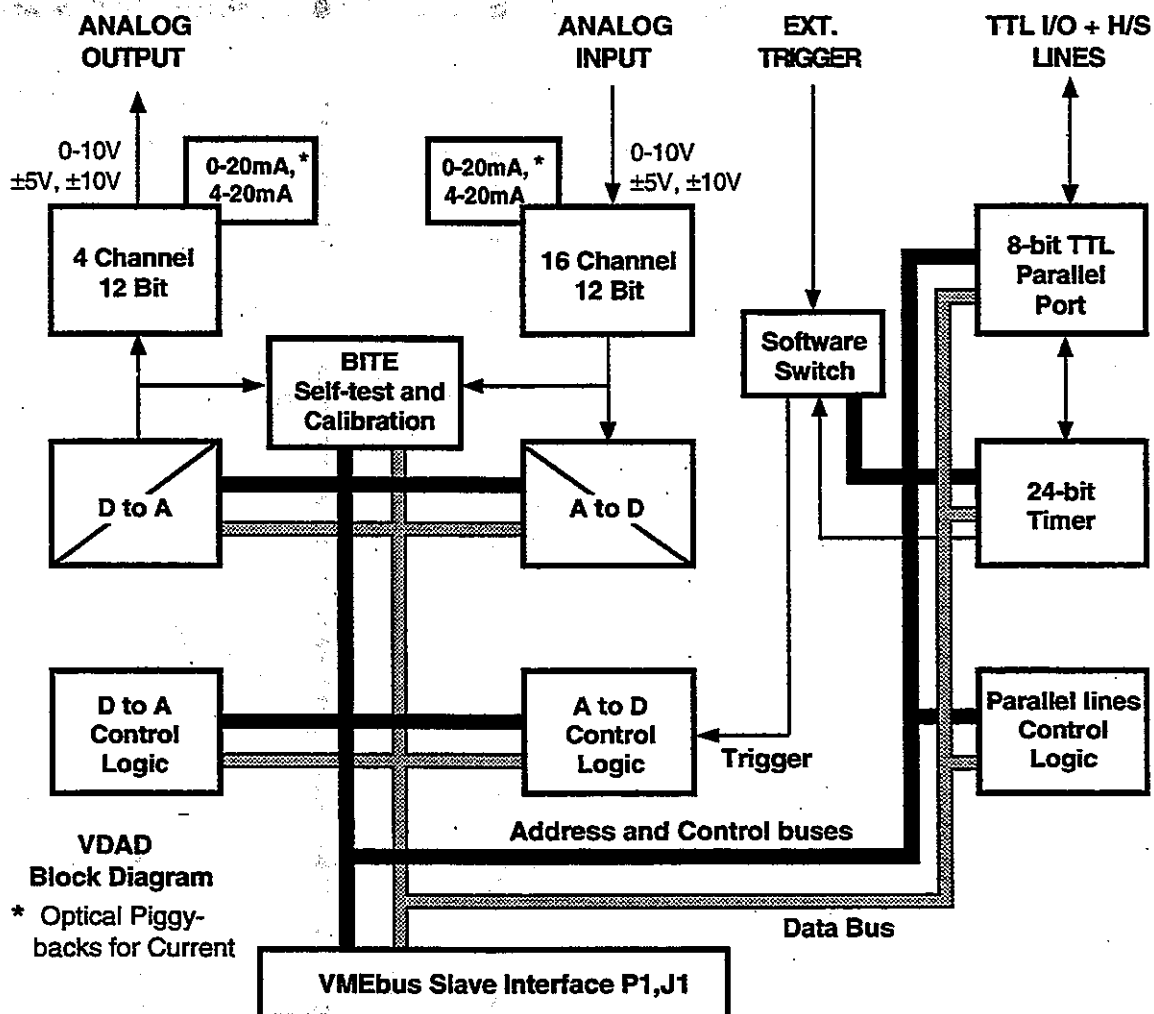
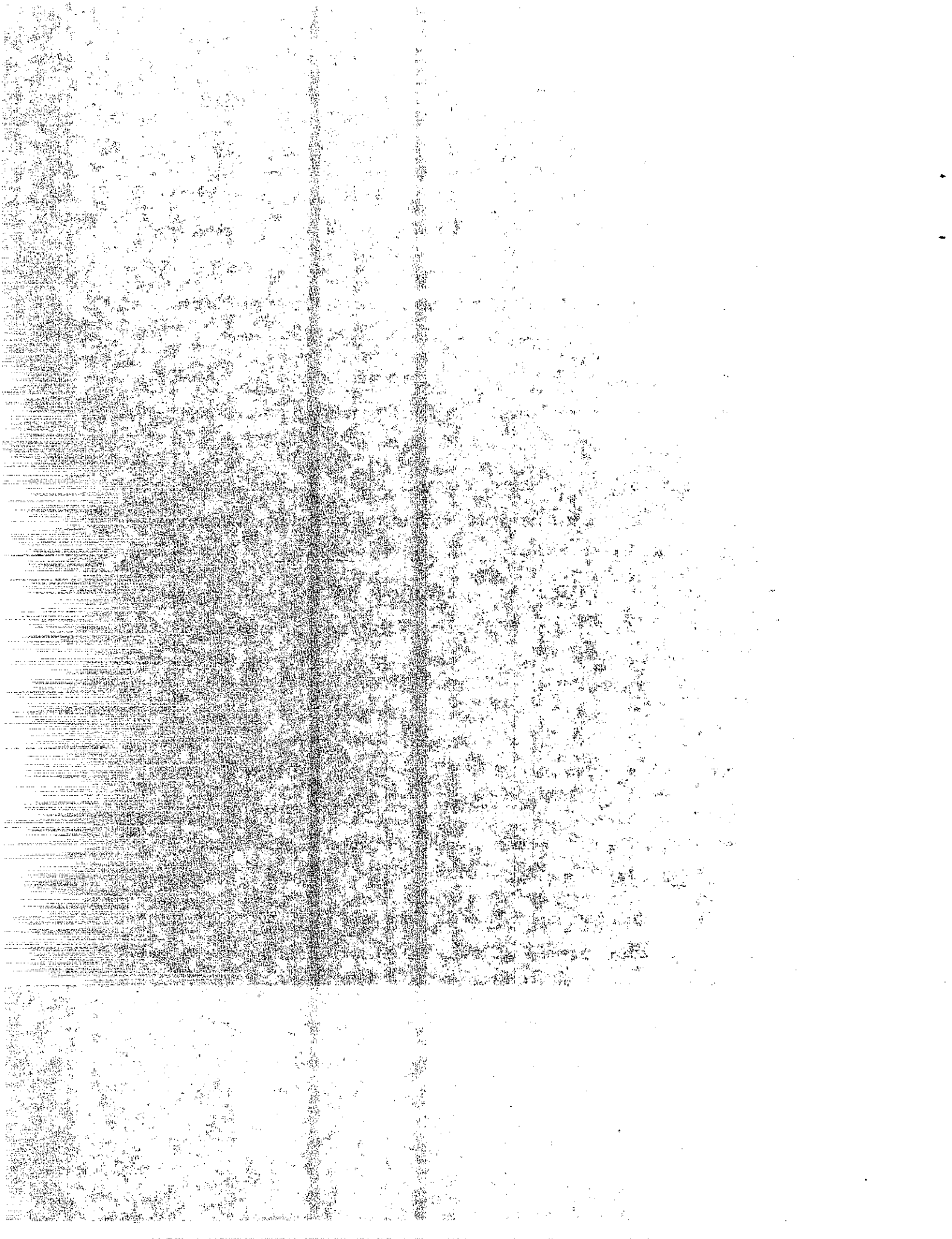


Fig. 11. **FUNCTIONAL BLOCK DIAGRAM OF ATC PROTOTYPE A/D MODULE**

4.1.1.4 Support Hardware

Support hardware includes the power supply, chassis, backplane termination, development system mass storage controller and mass storage devices, and peripheral hardware. The prototype field controller, as shown in Figures 4 and 5 on page 17, specifies a 19 inch rack mount chassis and a 15 slot backplane. The prototype development system, also shown in Figure 4, specifies a 6U desk top enclosure with a removable 19 inch subrack chassis, pedestal base with cooling fans, and polyurethane dust filters. This unit supports single height single backplane cards (P1 only), double height single backplane cards (P1 only), and expansion capabilities for double height double backplane cards (P1 and P2, see Figure 3). To prevent ringing, backplanes in both field controller and development systems utilize passive termination techniques (330 ohm resistor to +5V and 470 ohm resistor to ground at both ends of the bus). Passive termination increases power losses however, as continuous current in the terminating resistors creates an approximate 6 Watt drop. Whereas active termination peaks at approximately 6 Watts, but experiences a continuous drop of only 2-3 Watts. The 110V field controller power supply provides 50 watts at 4A/+5V, 2A/+12V, and 1A/-12V. The power supply front panel provides a toggle power switch and LED indicator for +5V, +12V, -5V, -12V, and Uopt (user option). Figures 12 through 15 illustrate the modular power supply design.

The prototype development system includes a 3.5 inch floppy disk drive, 40 MByte hard drive, and a mass storage controller module to support these devices. The mass storage controller module is a single height intelligent VME bus card capable of providing required control, formatting, and interface logic for 4 disk drives and 2 hard drives, or 3 disk drives, 1 tape drive, and 2 hard drives. The controller module, PEP Modular Computer's VMSC card, supports high speed transmission via direct access to the VME bus through a Z80 CPU and DMA, a 16 KByte EPROM, and a 16 KByte RAM chip set. Figure 16 provides a functional description of the mass storage controller module(9). Future systems may benefit from SCSI type controllers.



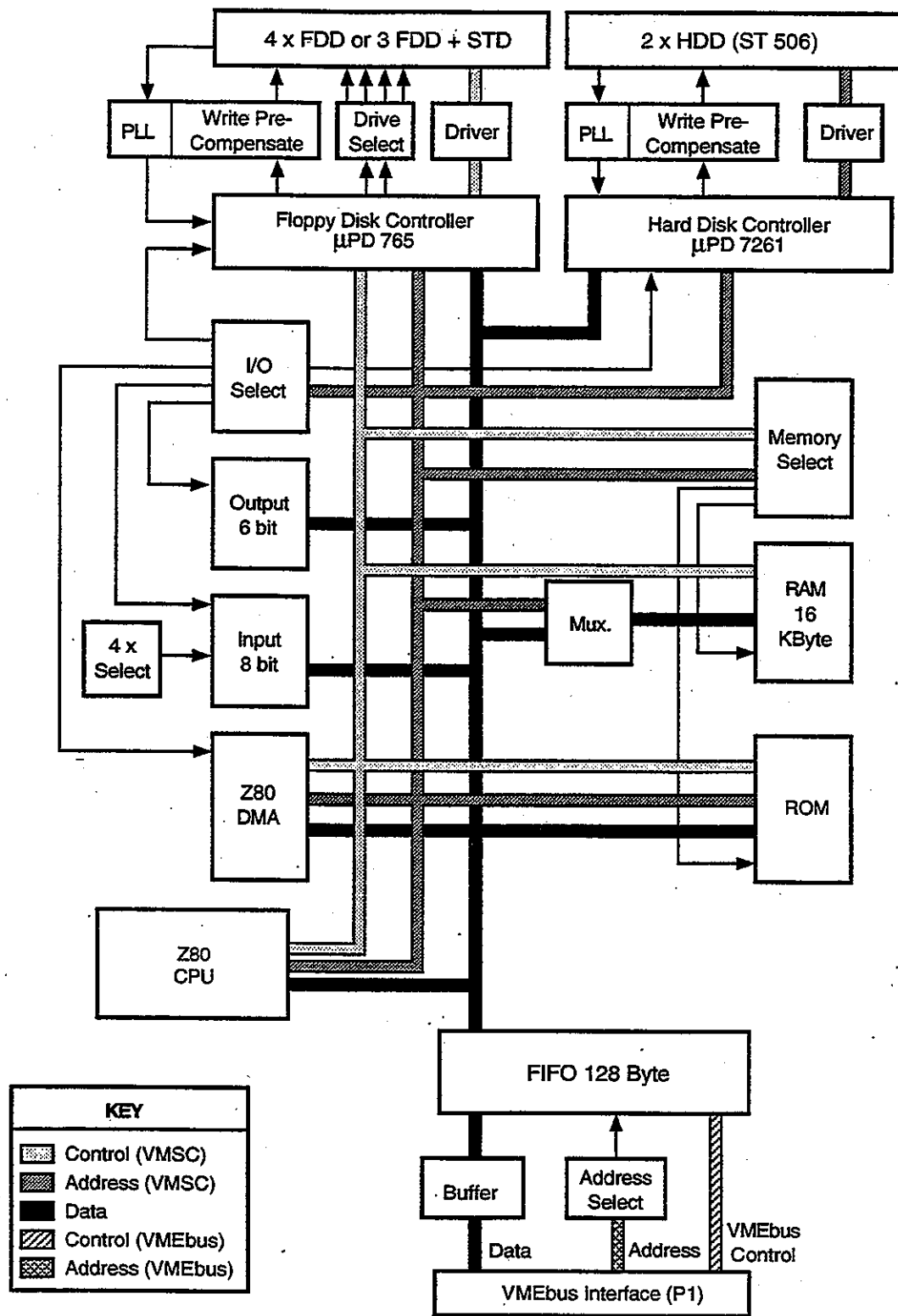


Fig. 16. FUNCTIONAL BLOCK DIAGRAM OF ATC PROTOTYPE MASS STORAGE CONTROLLER MODULE

4.1.2 Software

Caltrans operates thousands of microprocessor-based traffic controllers running solely on assembly code. Unfortunately, the complexity of modern system software and the lack of compilers, debuggers, and other development tools common to high-level languages and operating systems has resulted in complicated and lengthy routines providing very limited error checking and user interface capabilities.

High-level software however minimize these problems by providing features such as memory management, I/O control, compiling loading and execution management, file and directory control, user interface mechanisms, and multi-tasking capabilities. The ATC high level language and real-time operating system are discussed in the following sections. An object-oriented development package, written for Caltrans under a joint project by University of California Irvine, Carnegie Mellon University, and Louisiana State University is outlined in the following sections and detailed in Appendix B (2). Software specific to the ATC SWIM application is discussed in the Operational Test Section with source listings provided in Appendix D. ATC prototype operating system and programming language software is addressed below.

4.1.2.1 Programming Language

As stated previously, C is specified as the ATC prototype programming language. C, referred to as a mid-level language, supports high-level code, such as math, I/O, and system calls, and also supports low-level assembly code. C is a modular language based on independent functions promoting structured and reliable programming practices. It is an efficient language designed to produce compact routines that run relatively quickly. C is also a portable language. Routines can be developed on one platform and run on another with very little or no modification. Currently, C compilers are available for over forty different platforms (3).

Much of the SWIM application software was written in C. A Microsoft C test routine was developed to simulate a variety of digital axle sensor and analog weigh pad signals representing 15 different vehicle classes. All user interface and computational routines, weight, class, speed, and violation, were also developed in Microsoft C, and later ported to the VME bus ATC platform under OS-9TM C. Very little assembly code was required throughout the project with only a small portion needed for bit-level manipulation of simulation hardware and device drivers. Section 4.2.4 Operational Test, page 60, provides a description of the application software. A complete listing of the test routine and application code is found in Appendix D.

4.1.2.2 Operating System

OS-9TM, selected as the ATC real time operating system, is well suited for many transportation control applications. Run time kernels are small, efficient, and economical. Essential real time elements, such as preemptive task switching and reentrant and position-independent memory modules, allow for execution of a variety of interrupt driven functions with variable frequencies. OS-9TM includes independent file managers for many types of I/O, a fully ROMable kernel, development tools, and a multi-user environment promoting parallel software development. The ATC prototype field controller employs Industrial 68020/OS-9TM, a small real-time kernel designed for ROM-based applications requiring no disk or tape support. The prototype development system required Professional 68202/OS-9TM providing a programming environment with disk and tape support, a C compiler, an assembler, and an assembly debugger. Both Industrial and Professional packages are optimized for the 68020 CPU, while supporting 68000 software development. Additional support tools were purchased to aid in software development including: a system state debugger, a user state debugger, a C source level debugger, PCBRIDGETM a cross development tool supporting MS-DOSTM or OS-2TM based applications, assemblers, linkers, and a communication package.

SWIM application software employs OS-9'sTM real-time capabilities throughout the data acquisition process. A computational routine waits in a "sleep" state until a data-available signal is asserted. Interrupts from the loop detector (beginning of cycle indicator) and other interface circuitry signal sleeping axle sensor, weigh pad, and loop data-acquisition tasks to wake and begin execution. This process is described in detail later in the report. For more information on OS-9TM refer to Microware Systems Corporation(4).

4.1.2.3 Object-Oriented Application Development

The Advanced Transportation Controller Software (ATCS) package is an object oriented application generator designed to allow non-programmers to develop transportation software for the ATC prototype and similar VME bus platforms. This package is coded in C and OS-9TM. It is based on a library of pre-programmed functions that handle typical traffic engineering tasks. A number of Traffic Control BLocks (TCBLK) have been developed to count vehicles, measure occupancies, archive data, change metering rates and message sign warnings, cycle traffic signals, and execute other traffic engineering functions. Figure 17 illustrates the ATCS operating concept(2).

A complete application can be constructed by choosing TCBLKS from the function block library, defining block parameters (minimum/maximum values, active high/low etc.), and linking selected blocks to one another to form a complete strategy. These simple steps are accomplished through a user friendly interface with icons and pull down menus. A detailed description of the ATCS package is found in Appendix B.

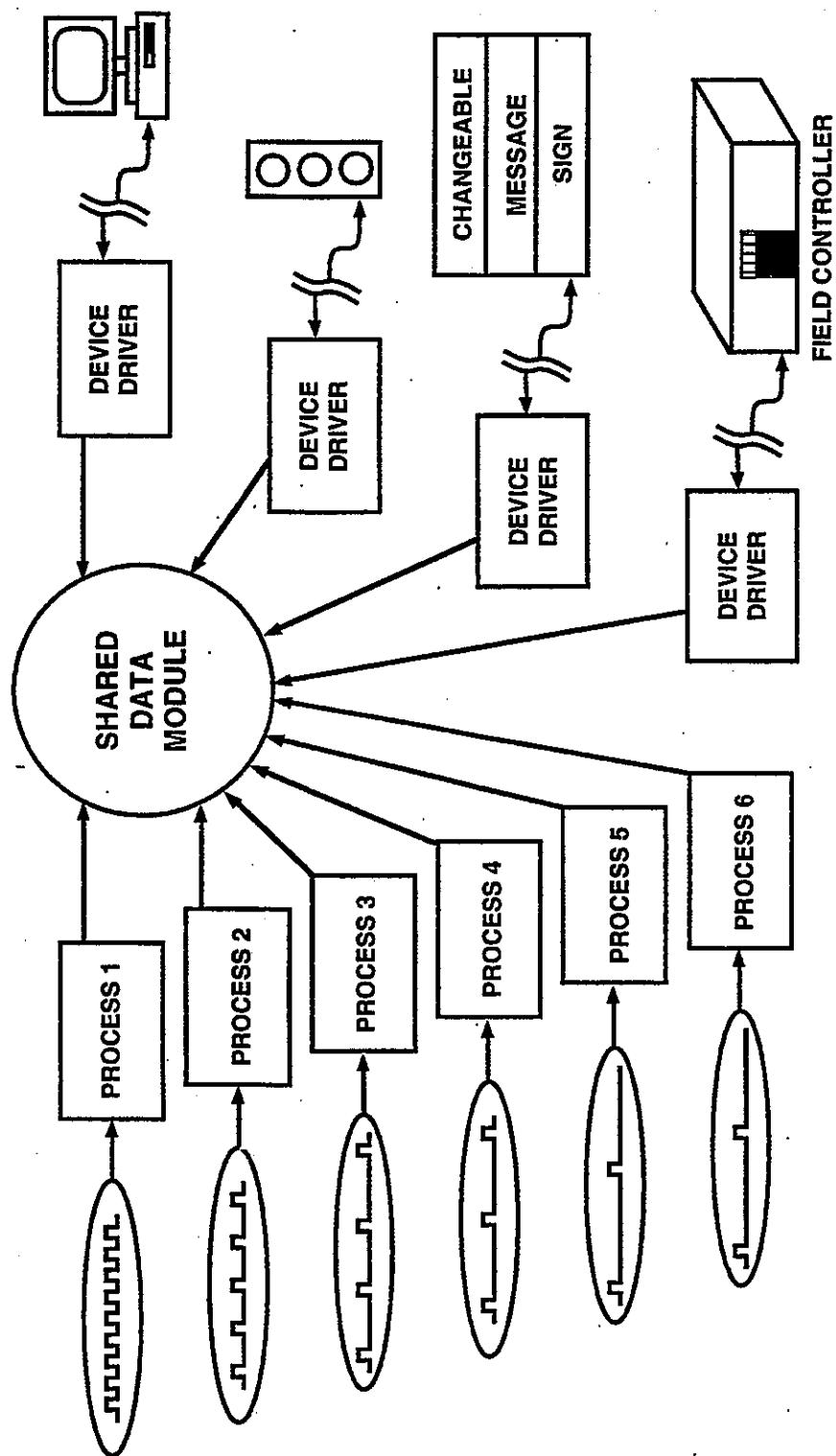


Fig. 17. OBJECT ORIENTED ATC SOFTWARE OPERATIONAL CONCEPT

4.1.3 Cost

During the course of this study two ATC controller units were purchased, one development system and one target system (field controller). Competitive bids were submitted for both systems from three manufacturers. The low-bid development unit, described previously, cost \$9746 (\$6778 hardware and \$2968 software) and the low-bid target unit cost \$6280 (\$6100 hardware and \$180 software). It should be noted, these are quantity one prices including special purpose cards and since the purchase of the systems, in 1990, costs have dropped substantially. Quotes in 1992 for a 68000 CPU-based target unit and a 68030 CPU-based target unit (both quantity 500) were \$1374 and \$2000 respectively. SWIM costs are detailed in later sections.

4.2 Slow Speed Weigh-in-Motion (SWIM) Prototype

Weigh-in-motion, as defined by the American Society for Testing and Materials (ASTM), is the "process of estimating a moving vehicle's gross weight and the portion of that weight that is carried by each wheel, axle, axle group, or combination thereof, by measurement and analysis of dynamic forces applied by its tires to a measuring device" (5). Highway Weigh-in-Motion (WIM) systems typically support one of three applications: i) collection of statistical traffic data for a variety of uses, ii) over-weight screening for use in truck weigh stations, and iii) over-weight enforcement for use in weigh stations. Number i) is achieved with high-speed (typically up to 80 mph) main-line WIM systems. Numbers ii) and iii) are accomplished with slow-speed (typically up to 40 mph) weigh station WIM systems. This prototype test provides over-weight screening (application ii). The ATC hardware and software, as described previously, collect and process data as vehicles pass over electronic sensors embedded flush across an off-ramp to a California Highway Patrol (CHP) weight station.

The data acquisition hardware, roadway sensors, interface electronics, and system software were designed to collect wheel loads, axle loads, axle group loads, gross vehicle weight, speed, center-

to-center axle spacing, axle group spacing, vehicle class (one of fifteen based on axle spacings and weight), site identification, date and time of vehicle record, sequential vehicle record number, and violation summary. The following sections describe test facilities, materials, installation procedures, and the operational concept. Appendix C provides details and data sheets pertaining to these topics. Appendix D provides source code listings for SWIM specific software.

4.2.1 Test Facilities

Both laboratory and field tests were conducted on the prototype system. Interface electronics were designed and data acquisition components were assembled at the laboratory facility. Simulation hardware and software was developed for pre-installation bench testing. The following two sections describe both field and laboratory test environments.

4.2.1.1 Laboratory

Caltrans Office of Traffic Improvement was established under Executive Order of the Governor of California in 1988 resulting in the creation of the Division of New Technology Materials and Research (DNTM&R). The Division's goal, to provide material testing, design assistance, and establish public/private partnerships implementing new technologies, traffic management strategies, and research to improve free-flow of traffic, is carried out through thirteen Offices. These offices include: Management Services, Advanced Systems Integration and Implementation, Advanced Transportation Management and Information Systems, Advanced Vehicle and Infrastructure Development, Electrical and Electronics Engineering, Research and Development Center, Rail Transit Aviation and Rural Technology, Engineering Geology, Engineering Liaison, Environmental and Engineering Services, Geotechnical Engineering, Pavement, and Structural Materials. Over 300 engineers, scientists, technicians, and support personnel are assigned to the 10 acre laboratory complex.

The Office of Electrical and Electronic Engineering (E³), responsible for the ATC project, conducts research and evaluates electrical and electronic systems for highway operations and maintenance. E³ facilities provide state-of-the-art tools such as oscilloscopes, logic analyzers, emulators, spectrum analyzers, protocol analyzers, PCB (printed circuit board) fabrication equipment, and CAE (computer aided engineering) packages for the design and development of electronic devices. E³ personnel researched, designed, developed, and installed the ATC SWIM system with assistance from the Office of Structural Materials - Machine shop, Headquarters Office of Traffic Operations - Electrical Systems, and Caltrans District 03 - Maintenance.

4.2.1.2 Field Site

Site selection was based on a number of factors including proximity to the DNTM&R laboratory, geometrical configuration, and availability of existing SWIM hardware. State operated SWIM locations in Sacramento, Castaic, Cordelia, Gilroy, Livermore, Los Banos, and Truckee were evaluated for ATC prototype field testing. SWIM hardware at Castaic and Cordelia had been removed. The Gilroy facility was still under construction and not yet operational. The geometry of the Livermore site had proved to create back-up problems in the past and was deemed unsuitable for test purposes. The Los Banos and Truckee facilities were under repair at the time of site selection. The Sacramento (Antelope Rd.) site was not operational, as it was installed for test purposes some 10 years ago and was never intended for on-going SWIM operations. Based on these findings, a decision was made to utilize the existing equipment cabinet and install new SWIM sensors at the Antelope facility located off Interstate 80, less than 20 miles from the DNTM&R laboratory.

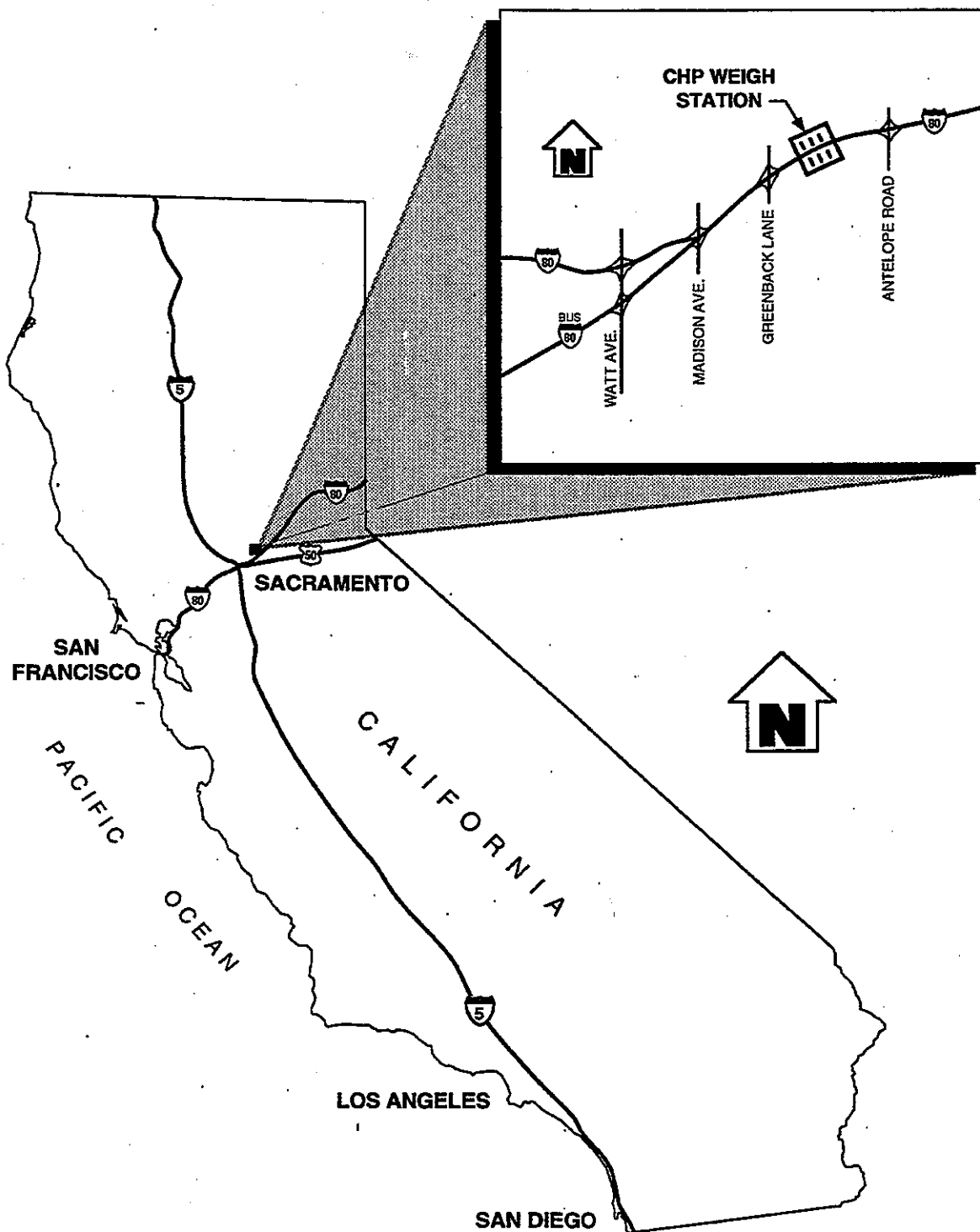


Fig. 18. SACRAMENTO AREA MAP WITH I-80 / ANTELOPE ROAD WIM STATION BLOW-UP

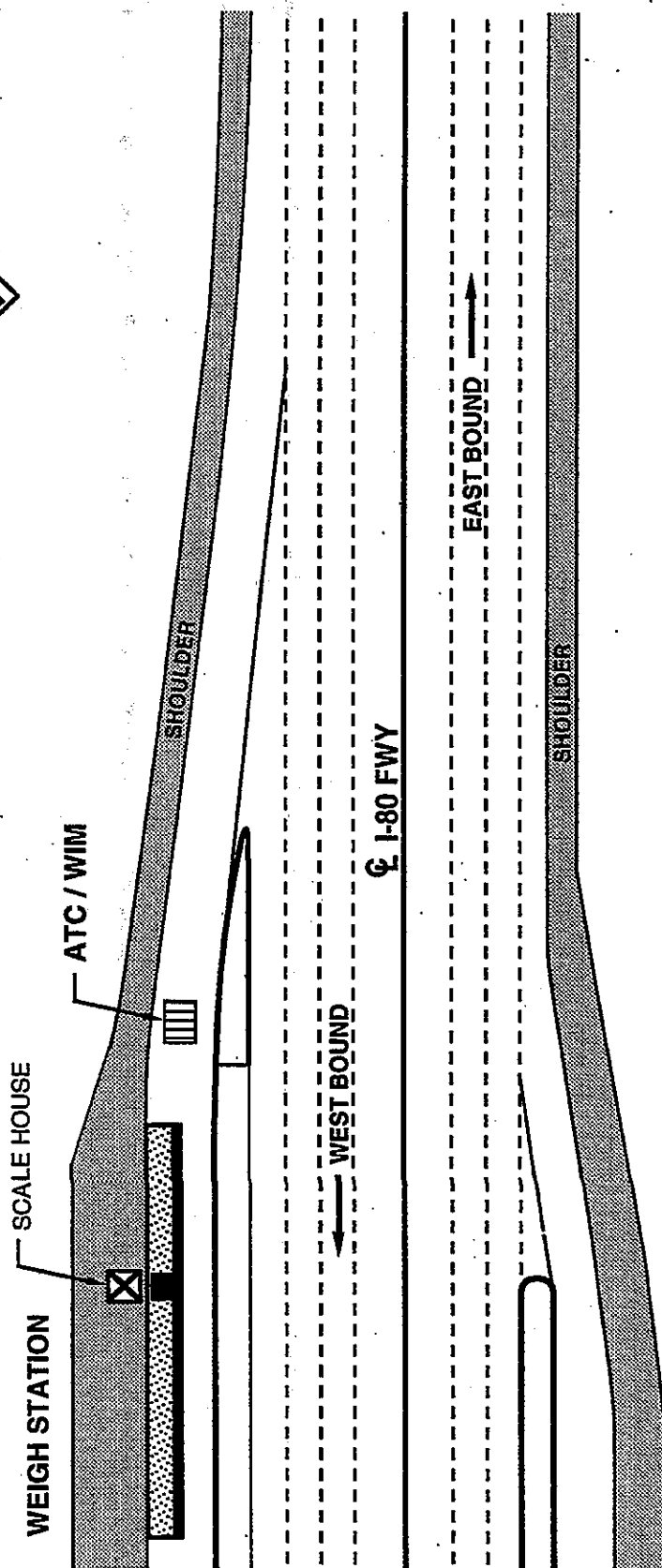


Fig. 19. I-80 / ANTELOPE ROAD WEIGH STATION

Site Location (Feet From Gore Point)	Mean Speed (mph)	Mode Speed (mph)	Medium Speed (mph)
400	37	31	37
500	35	44, 26	39
600	32	33	31
700	30	30	30

Table 1. ANTELOPE ATC SWIM SITE VEHICLE SPEEDS

The ATC SWIM prototype system was installed in the Westbound CHP weigh station off-ramp as shown in Figures 18 and 19. The Antelope site experiences a typical daily traffic volume of 800 to 1000 trucks. The weigh station off-ramp, measuring approximately 1300 feet in length from painted gore point to static weigh scale, experiences approximately three percent cross slope. The single lane concrete off-ramp is twelve feet wide with eight foot asphalt shoulders. Adjacent to the asphalt shoulder is an additional dirt shoulder approximately twenty feet in width. The SWIM roadway components were installed approximately 600 feet from the gore point of the off-ramp. This point was selected to provide the trucks with over 500 feet of lane change and stopping distance before reaching the static scale and enough distance from the gore point to avoid average speeds exceeding the maximum 40 mph SWIM speed. Table 1 summarizes vehicle speeds at four locations.

4.2.2 Test Hardware

SWIM sensors include an inductive loop detector to indicate the presence of a vehicle and in turn initiate the data acquisition process, a series of eleven axle sensors to measure axle spacings and in turn produce vehicle class types, and two weigh pads to sense wheel loads and in turn provide vehicle weights. Data acquisition circuitry includes the ATC field controller, signal conditioning and sensor interface hardware, and peak-hold interrupt electronics. An operational description of the complete SWIM system is provided in the Operational Test Section 4.2.4, page 57. SWIM sensors and data acquisition hardware are described below. Data sheets and detailed installation procedures are found in Appendix C.

4.2.2.1 ATC Data Acquisition Circuitry

The data acquisition cabinet is configured as illustrated in Figure 20. A standard Input File assembly, as described in the Model 170 Traffic Signal Control Equipment Specifications (TSCES) was modified to accommodate horizontally mounted axle sensor interface cards, vertically mounted weigh pad interface cards, vertically mounted peak hold detector cards, and a

vertically mounted loop detector module. This configuration supports easy access and break-out of the many multi-conductor sensors channels.

A block diagram illustrating the data flow between devices housed in the input file is shown in Figure 21. A signal from the inductive loop triggers the inductive loop detector which passes a presence signal to the digital input card which in turn interrupts ATC controller processing, indicating the presence of a vehicle and a new vehicle record. Digital data from axle sensors are sensed and conditioned at the axle sensor interface cards and passed to the digital input cards and on to the controller CPU module. Analog data from weigh pads are sensed and conditioned at the weigh pad amplifier cards, amplified by the differential amplifiers, and passed to the Peak Hold Detector cards. The Peak Hold Detectors capture and hold the maximum amplitude of the positive analog voltage signal. Once the maximum is obtained, it is passed to the A/D card where it is digitized. A trigger, asserted by the digital input card, interrupts controller processing requesting a read and write of the maximum voltage, now in digital form. Once the controller has stored the voltage, it asserts a Peak Hold reset signal, again through the digital input/output card.

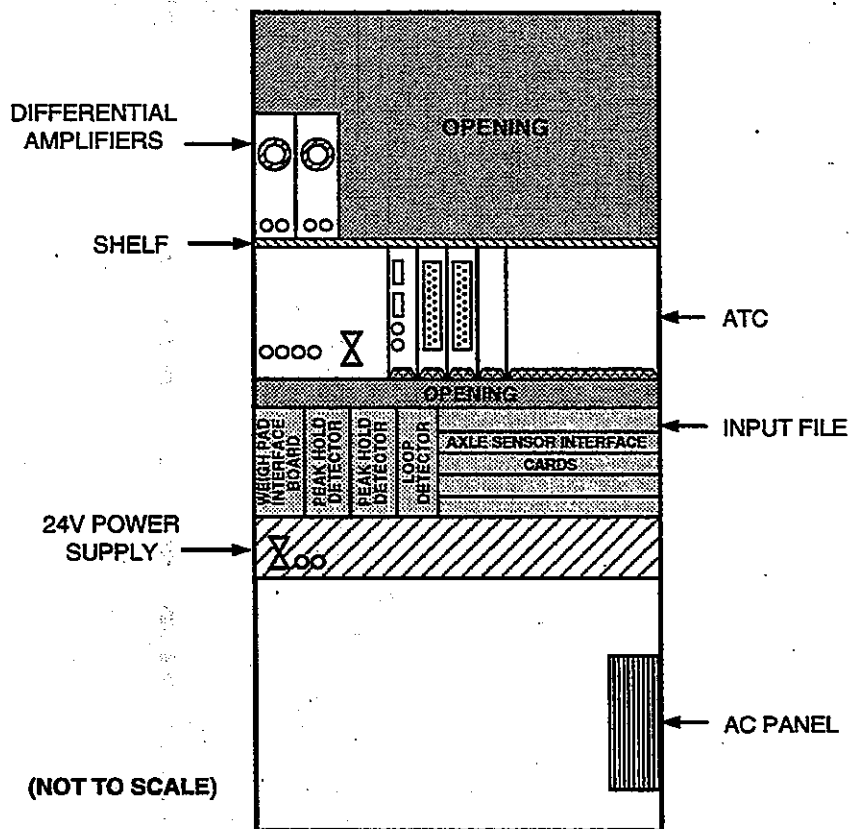


Fig. 20. ATC SWIM SYSTEM CABINET CONFIGURATION

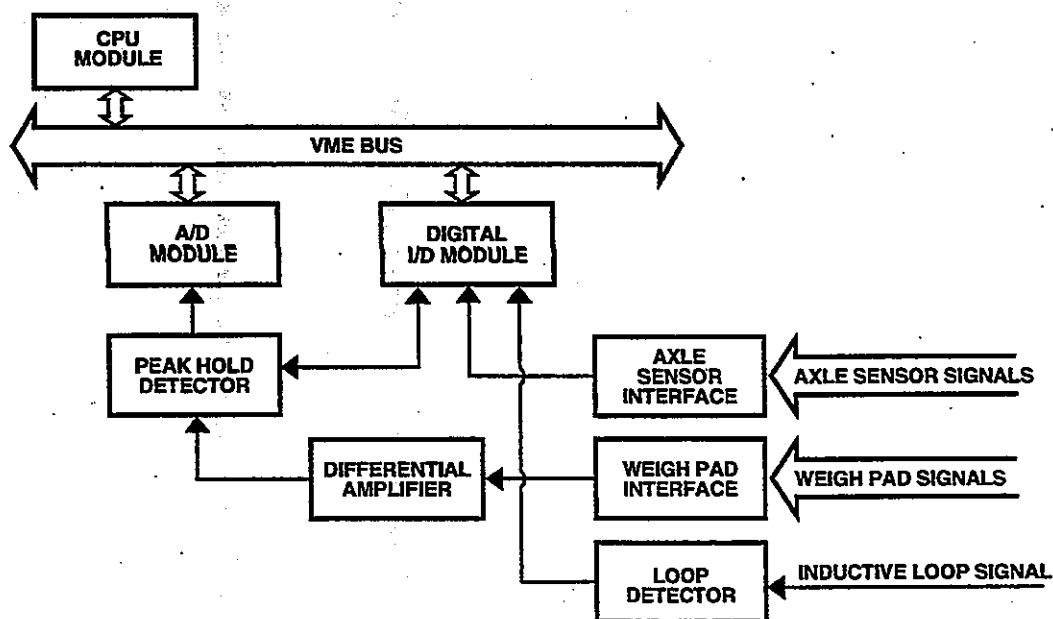
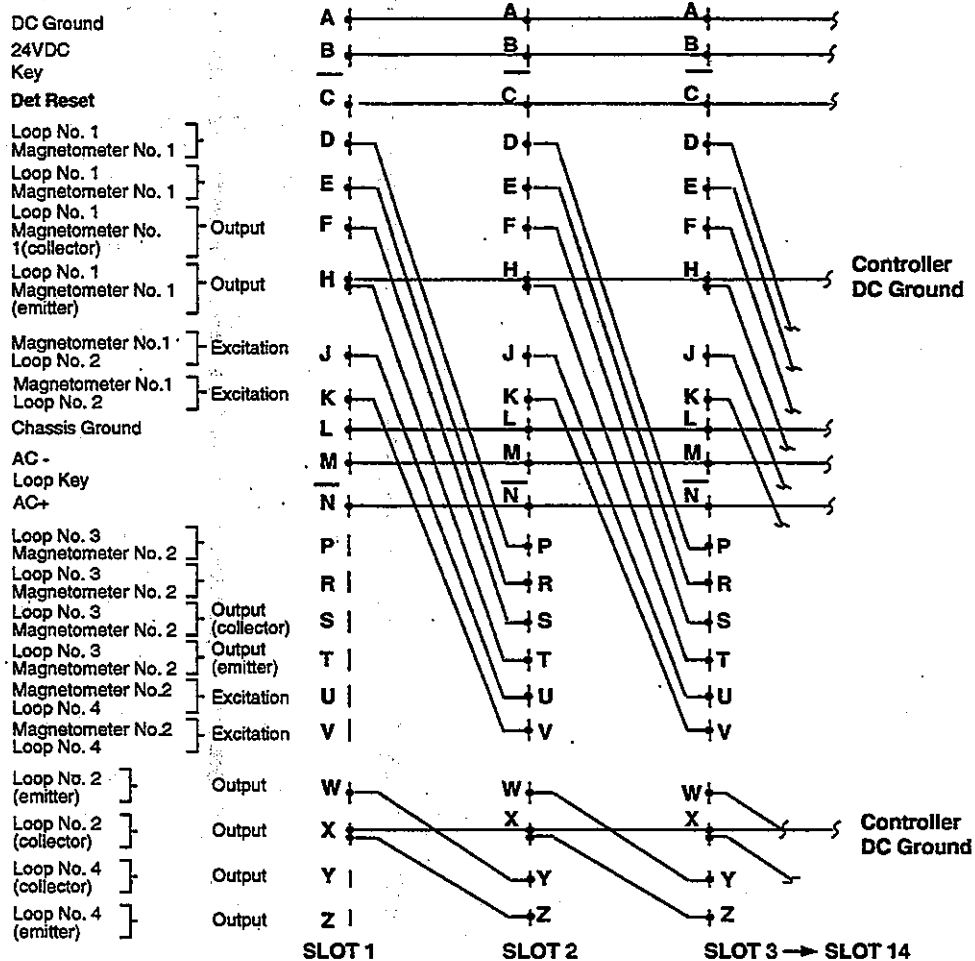


Fig. 21 ATC SWIM SYSTEM DATA FLOW DIAGRAM

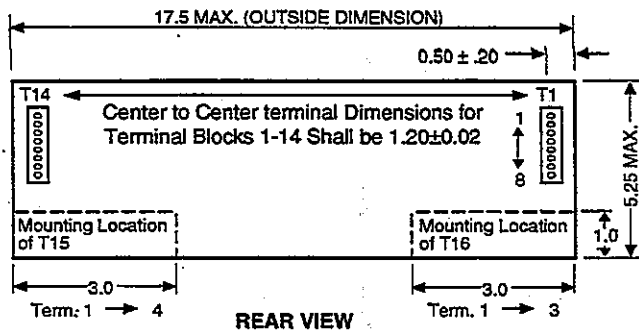
Figure 22 shows the standard Traffic Signal Control Equipment Specification (TSCES) Input File. Slot/terminal 1 was modified to accommodate the weigh pad amplifier card pin-in-socket connector (TSCES calls for edge connectors). Slot/terminal 2 is unused. Terminal 3, corresponding to the loop detector slot, was physically disconnected from the rest of the backplane which remains as defined in the TSCES detail. Slots 4 and 5 are occupied by Peak Hold Detector boards and slots 6 through 14 support axle sensor interface cards. Pins F, W, D, E, and J on all terminals remain available for independent card operation. Terminal bus L is tied to equipment ground, terminal bus N is tied to +12 VDC, and terminal bus M is tied to analog ground.

Table 2 identifies the Input File wiring scheme for the Peak Hold Detector boards. Used in conjunction with a weigh pad and weigh pad amplifier board, the Peak Hold Detector will capture the maximum weight applied by any single wheel of a vehicle. The peak hold detector performs three unique operations, each dependent on the other as shown in Figure 23. The operations occur sequentially starting with the peak hold circuit capturing the maximum voltage, followed by the trigger circuit interrupting the ATC controller, and finally by the reset circuit resetting the peak hold circuit. This process occurs for each wheel applied to the weigh pad. Figure 24, acquired with a LeCroy Model 9400 Dual Channel 125 MHz Digital Oscilloscope, shows the input to the peak hold (channel 1), with the peak and reset points indicated by arrows and the output of the peak hold circuit (channel 2), again with the peak and reset points indicated by arrows. A detailed description of the peak hold detector is found in Appendix C.

INPUT FILE WIRING DIAGRAM



INPUT FILE DETAIL



INPUT FILE TERMINAL ASSIGNMENT DETAIL

T1-14		T15		T16	
Term	Pin - Function	Term	Function	Term	Function
1	SP - Spare	1	+24VDC	1	AC+
2	F - Channel 1 Output	2	DC Ground	2	AC-
3	W - Channel 2 Output	3	Det Reset	3	Equip. Ground
4	D - Channel 1 Input	4	C1 Harness DC Ground		
5	E - Channel 1 Input				
6	J - Channel 2 Input				
7	K - Channel 2 Input				
8	L - Equip. Ground				

Fig. 22. CALIFORNIA DEPARTMENT OF TRANSPORTATION TRAFFIC SIGNAL EQUIPMENT SPECIFICATION INPUT FILE

Terminal		Board Pin #	Input	
Strip #	Pin #		Weigh Pad #	Function
4	SP	N	1	+12 VDC
4	F	F	1	Reset (input)
4	W	W	1	Trigger (out)
4	D	D	1	Output (analog)
4	E	E	1	Input (analog)
4	J	J	1	+5VDC
4	K	K	1	-12 VDC
4	L	L	1	Ground
4		M	1	Analog Ground
5	SP	N	2	+12 VDC
5	F	F	2	Reset (input)
5	W	W	2	Trigger (out)
5	D	D	2	Output (analog)
5	E	E	2	Input (analog)
5	J	J	2	+5 VDC
5	K	K	2	-12 VDC
5	L	L	2	Ground
5		M	2	Analog Ground

Table 2. **PEAK HOLD DETECTOR BOARD INPUT FILE WIRING**

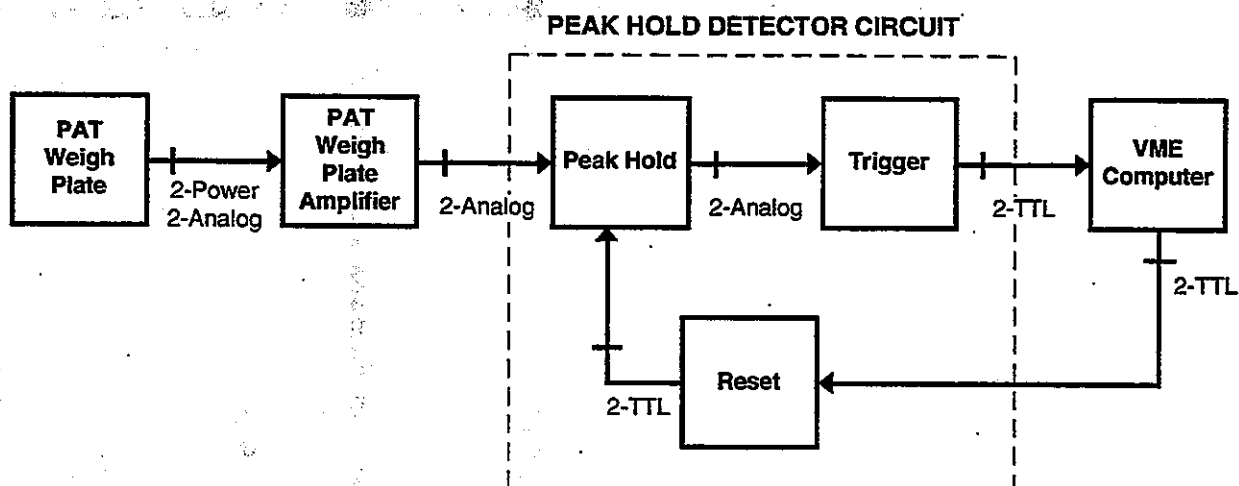


Fig. 23. PEAK HOLD DETECTOR BLOCK DIAGRAM

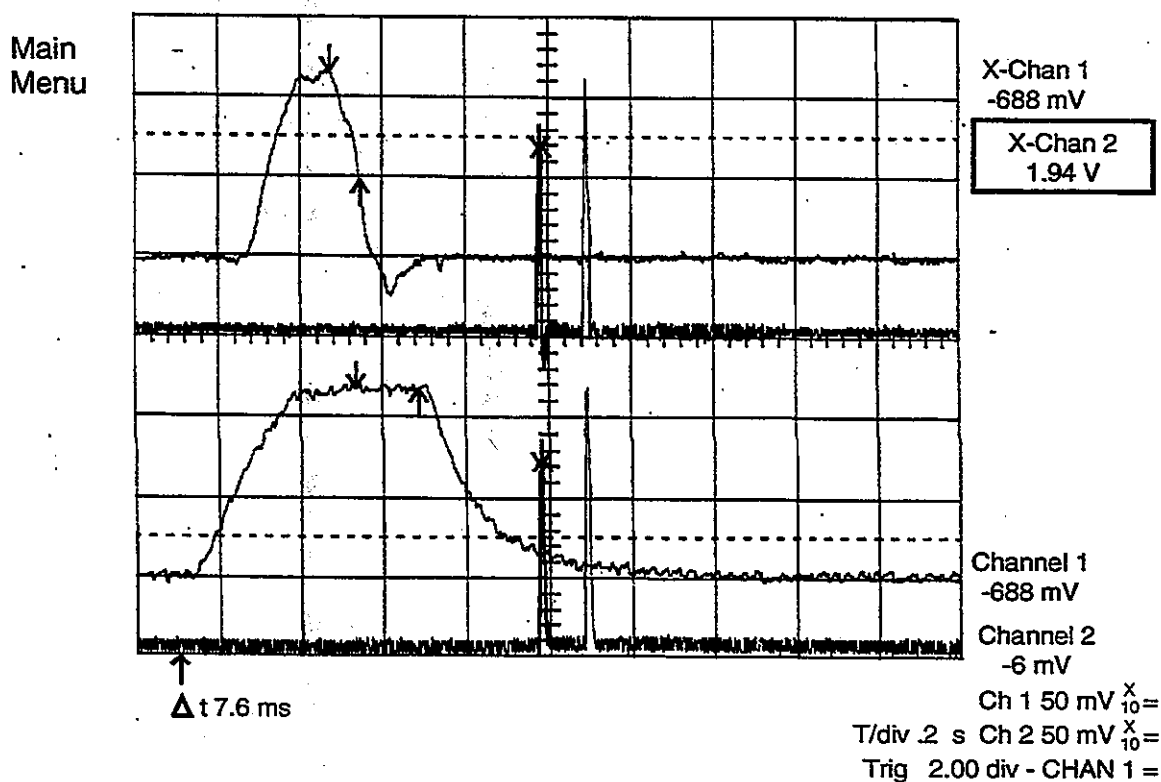


Fig. 24. PEAK HOLD DETECTOR INPUT SIGNAL (CHANNEL 1)
AND OUTPUT SIGNAL (CHANNEL 2)

As shown in Figure 21, a differential amplifier was placed between the output of the weigh pad amplifier cards and the input of the peak hold detector cards. These amplifiers, Hewlett Packard Model 2470A Data Amplifiers, provide wide-band, high-gain differential amplification of low-level signals. Typical signal sources for the devices are resistive transducers with output resistances of 1000 ohms or less. Maximum input and output levels are ± 11 V and ± 10 V respectively. These devices, with several fixed gain positions and 10-turn verniers, were set at a $\times 10$ amplification factor for amplification and buffering between the differential weigh pad interface card and the single-ended peak hold detector board. As mentioned previously, a new peak hold detector design includes additional filtering and differential amplification, effectively eliminating the need for the HP amplifiers.

4.2.2.2 Weigh Pads

The prototype SWIM system weigh pad instrumentation includes 2 bending plate weigh pads and a preamplifier interface card. The weigh pad assemblies, provided by PAT Equipment Corporation, Inc., consist of rectangular 20 inch by 69 inch steel plates one inch in depth, as shown in Figure 25 (11). Each plate weighs 265 pounds and is supported lengthwise along the edges by a metal frame. The rectangular steel plate is instrumented with strain gauges and when supplied with an excitation voltage produces a strain voltage which is applied to the preamplifier interface card. The axle weight range supported by each weigh pad is 1,100 to 44,000 pounds, with a 10,000 pound load corresponding to a 1 V output signal. The rectangular steel weigh plate, strain gauges, and wiring are encapsulated with vulcanized synthetic rubber to protect the assembly from moisture and physical damage. Weigh pad assemblies compensate for temperature variations and operate from -50° F to 176° F. Each weigh pad and preamplifier interface card require a 12 VDC to 15 VDC regulated supply.

Two weigh pads, one per wheel, provide wheel weights and when summed produce axle weights. The weight pad interface card supports six differential weigh pad channels, two differential inputs

to determine Wheatstone Bridge (strain gauge) supply levels, and an additional four user defined channels. Each weigh pad channel provides a measuring amplifier containing a third order filter, a differential amplifier providing an overall gain of 200, and zero tracking electronics to compensate for detector zero fluctuations. Two summers are also provided to produce composite weight signals (axle weights vs. wheel weights). The interface card, supplied by PAT Equipment Corporation, Inc., measures 160 mm by 100 mm by 19.5 mm, operates from -20°C to 70°C , and draws 36 mW of power. Figure 26 illustrates the preamplifier interface card operational concept (11). Figure 27 illustrates the output signal of the weigh pad interface card where channel 1 shows a five axle truck with a steering axle and two sets of tandem axles and channel 2 magnifies the steering axle reading. Table 3 summarizes the interface card input file wiring.

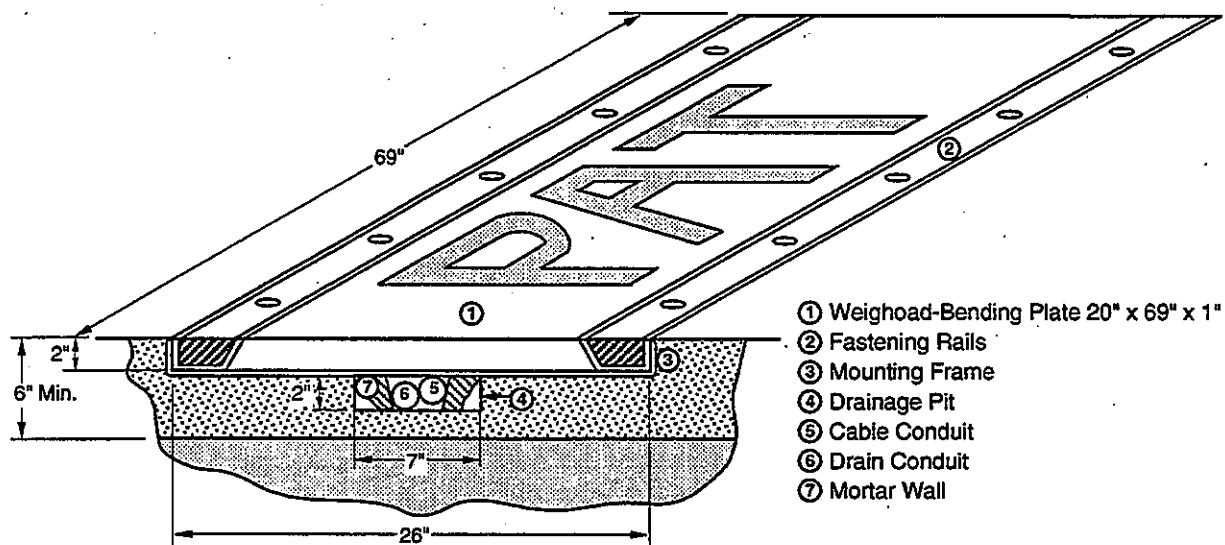


Fig. 25. PAT EQUIPMENT CORPORATION, INC.,
WEIGH PAD ASSEMBLY

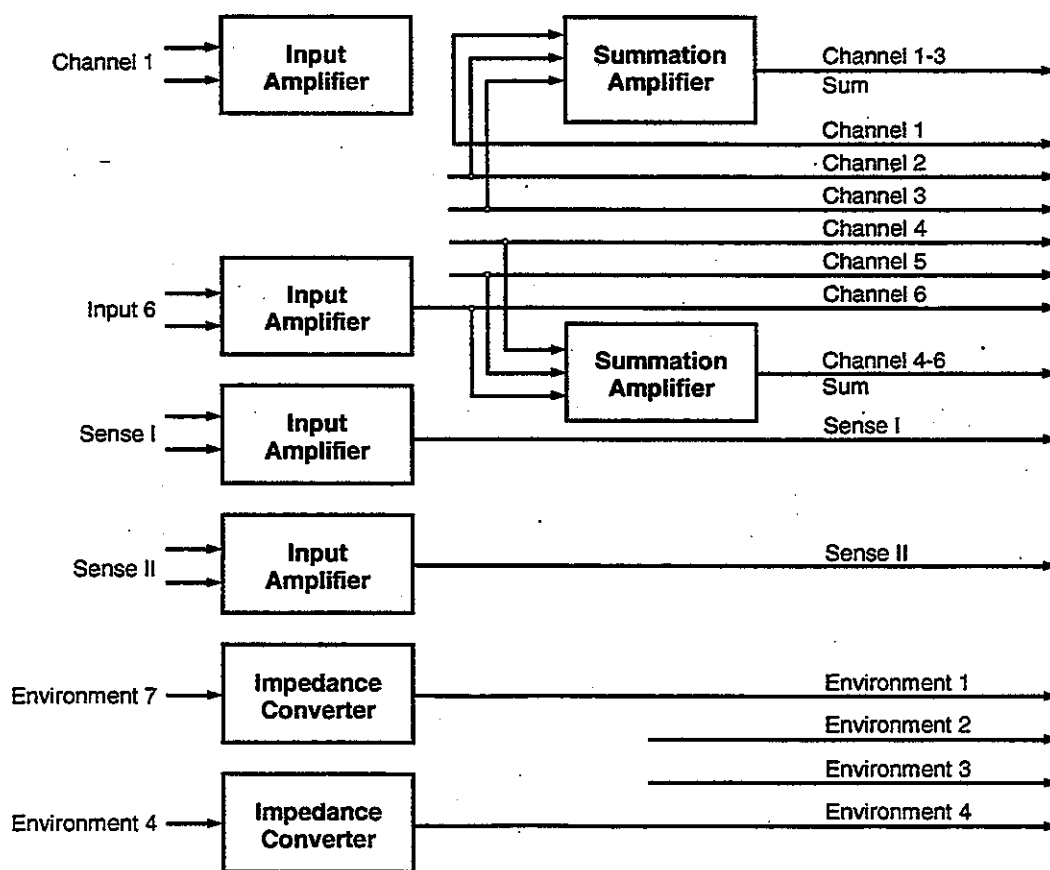


Fig. 26. PAT EQUIPMENT CORPORATION, INC., WEIGH PAD
PREAMPLIFIER INTERFACE CARD BLOCK DIAGRAM

Main
Menu

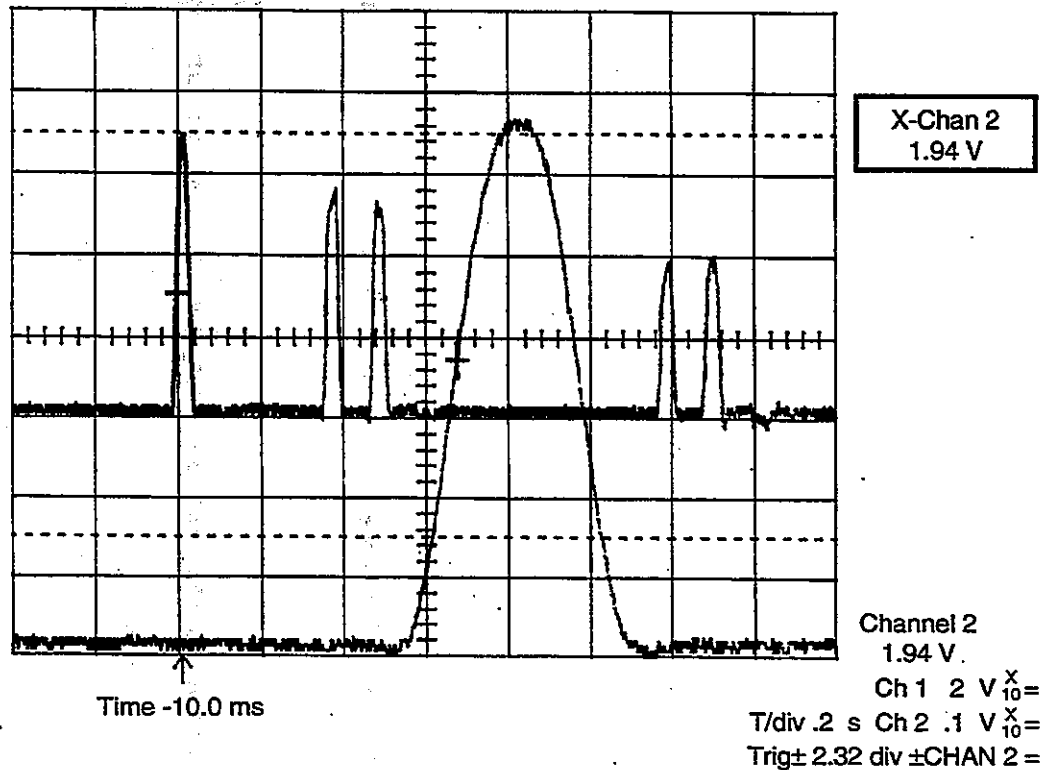


Fig. 27. FIVE AXLE TRUCK WEIGH PAD OUTPUT SIGNAL (CHANNEL 1)
AND EXPANDED WEIGH PAD OUTPUT SIGNAL (CHANNEL 2)

Terminal Strip # / Pin #	Board Pin #	Input	
		Weigh Pad #	Function
1 - SP	1a	1	+12 VDC (E+ wht / blu)
	1c	2	+12 VDC (E+ wht / org)
1 - F	8a	2	Input (S+ wht / brn)
1 - W	8c	2	Input (S- brn / wht)
1 - D	7c	1	Output (to amp)
1 - E	6a	1	Input (S+ grn / wht)
1 - J	6c	1	Input (S- wht / grn)
1 - K	9c	2	Output (to amp)
1 - L	31a	1	Ground (E- blu / wht)
1	31c	2	Ground (E- org / wht)

Table 3. WEIGH PAD AMPLIFIER INTERFACE BOARD INPUT FILE

4.2.2.3 Axle Sensors

The prototype SWIM system axle sensor instrumentation consists of eleven International Road Dynamic Inc. (IRD) Dynax Model 406 Replaceable Axle Sensors, and three axle sensor interface cards. Opposed to high-speed WIM, SWIM applications cannot assume constant vehicle velocities (trucks in weigh station facilities are typically decelerating), therefore, multiple axle sensors placed in a strategic configuration, described later, are used to acquire dynamic speeds resulting in accurate axle spacing measurements and in turn accurate vehicle classification.

The axle sensors are resistive type sensors mounted in a rectangular steel frame, approximately 6 inches x 2 inches x 100 inches, supported lengthwise along the edges by a pair of metal clamps allowing easy replacement after installation in the roadway. The sensor material is resistive in nature, where the application of pressure from a wheel causes a decrease in resistance between 2K ohms and 50K ohms. Under no-load conditions the resistance is greater than 10M ohms. The sensors, measuring 1 inch by 1 inch by 8 feet and 1/2 inch, are enclosed in a semi-rigid rubber material to protect the device from moisture and physical damage. Figure 28 illustrates the axle sensor assembly (10).

The eleven axle sensors are interfaced to the data acquisition system through 3 four channel Dynax Interface cards. The cards provide power for the main interface and optocoupler output circuitry. Supply power can range from 5 VDC to 18 VDC and in the SWIM system is provided by the +12VDC VME bus supply. Each sensor input channel provides potentiometers for control of trigger thresholds. The pots can be adjusted to set weight threshold trigger levels (i.e. the interface can be adjusted so a signal will not be produced unless a vehicle with a given threshold weight passes over the sensor). LEDs indicate the active-low axle hits. Figure 29 shows the output of the axle sensor interface card as a five axle vehicle passes over the sensors. Table 4 summarizes axles sensor interface board input file wiring.

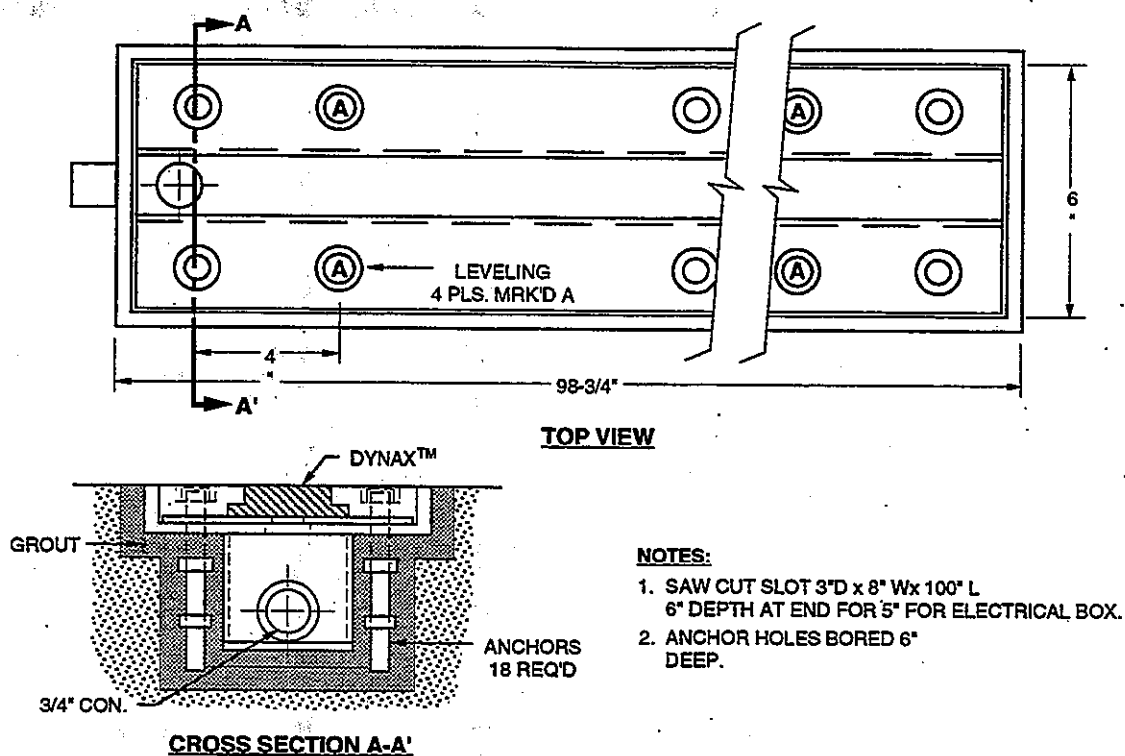


Fig. 28. INTERNATIONAL ROAD DYNAMICS, INC., REPLACEABLE DYNAX AXLE SENSOR ASSEMBLY

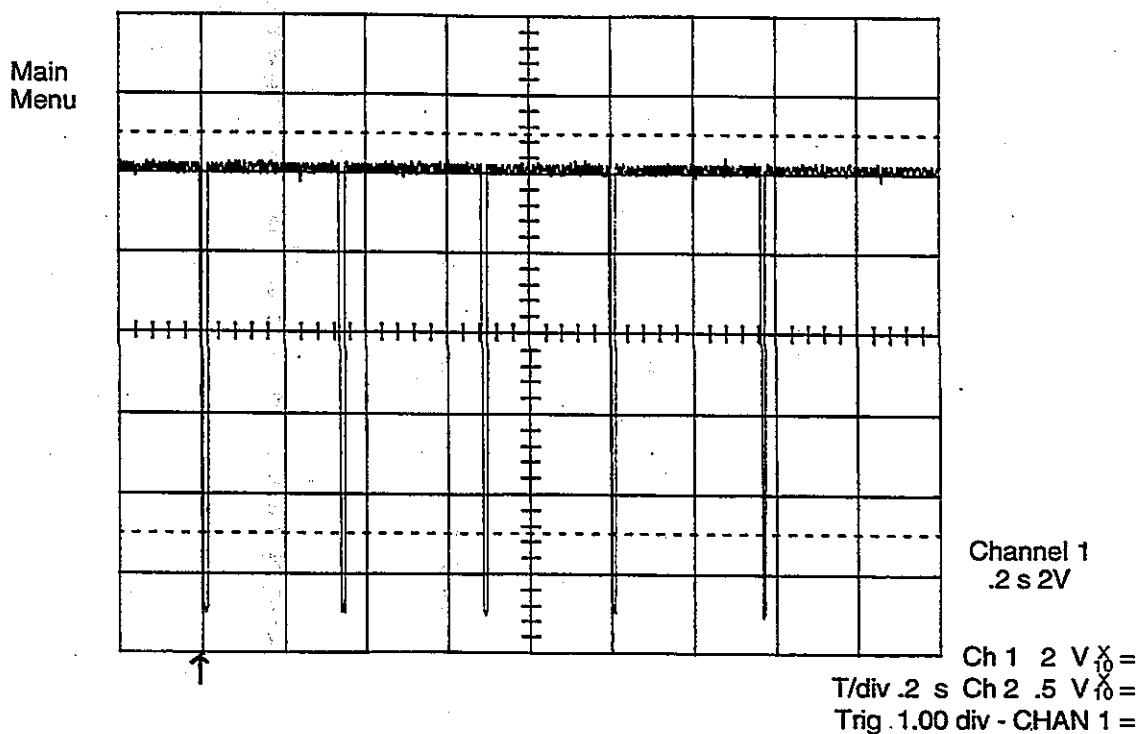


Fig. 29. INTERNATIONAL ROAD DYNAMICS, INC., REPLACEABLE DYNAX AXLE SENSOR OUTPUT SIGNAL

Terminal Strip # / Pin #	Board # / Pin #	Sensor Int # / Pin #	Sensor # / Function / Color
6 / 1 - SP 2 - F 3 - W 4 - D 5 - E 6 - J 7 - K 8 - L	No Connection 1 / S Y P R U V No Connection	1 / J5 - 1 J5 - 2 J3 - 1 J5 - 3 J5 - 4 J3 - 2	SO / In / Blu In / Blk Out / Wht * S1 / In / Grn In / Blk Out / Blk *
7 / 1 - SP 2 - F 3 - W 4 - D 5 - E 6 - J 7 - K 8 - L	1 / N F W D E J K L	1 / J2 - 2 J5 - 5 J5 - 6 J3 - 3 J5 - 7 J5 - 8 J3 - 4 J2 - 1	+12V S2 / In / Yel In / Blk Out / Yel * S3 / In / Brn In / Blk Out / Blk * Gnd
10 / 1 - SP 2 - F 3 - W 4 - D 5 - E 6 - J 7 - K 8 - L	No Connection 2 / S Y P R U V No Connection	2 / J5 - 1 J5 - 2 J3 - 1 J5 - 3 J5 - 4 J3 - 2	S4 / In / Blu In / Blk Out / Wht * S5 / In / Grn In / Blk Out / Blk *
11 / 1 - SP 2 - F 3 - W 4 - D 5 - E 6 - J 7 - K 8 - L	2 / N F W D E J K L	2 / J2 - 2 J5 - 5 J5 - 6 J3 - 3 J5 - 7 J5 - 8 J3 - 4 J2 - 1	+12V S6 / In / Yel In / Blk Out / Yel * S7 / In / Brn In / Blk Out / Blk * Gnd

Note: * Black-Wht and Blk-Yel outputs are wire pairs.

continues ...

Table 4. AXLE SENSOR INTERFACE BOARDS INPUT FILE WIRING

... Con't

Terminal Strip # / Pin #	Board # / Pin #	Sensor Int # / Pin #	Sensor # / Function / Color
13 / 1 - SP 2 - F 3 - W 4 - D 5 - E 6 - J 7 - K 8 - L	No Connection 3 / S Y P R U V No Connection	3 / J5 - 1 J5 - 2 J3 - 1 J5 - 3 J5 - 4 J3 - 2	S8 / In / Blu In / Blk Out / Wht * S9 / In / Grn In / Blk Out / Blk *
14 / 1 - SP 2 - F 3 - W 4 - D 5 - E 6 - J 7 - K 8 - L	3 / N F W D E J K L	3 / J2 - 2 J5 - 5 J5 - 6 J3 - 3 J5 - 7 J5 - 8 J3 - 4 J2 - 1	+12V S10 / In / Yel In / Blk Out / Yel * Spare / In / Brn In / Blk Out / Blk * Gnd

Note: * Black-Wht and Blk-Yel outputs are wire pairs.

Table 4. **AXLE SENSOR INTERFACE BOARDS INPUT FILE WIRING**

4.2.2.4 Loops

A 6 foot by 6 foot inductive loop, using four windings of #12 AWG stranded loop conductor wire embedded in the roadway, signals the presence or absence of a vehicle. This configuration, referred to as "Type A" in the TSCES, is the most common loop configuration for vehicle detection at signalized intersections and for occupancy and speed measurements on freeways. When a signal is applied to the loop, a passing metal mass (vehicle) alters associated parameters of inductance, resistance, capacitance and Q (quality factor of the resonant circuit). A Detector Systems, Inc., Model 222B Two Channel Loop Detector, (12), monitors the loop channel for rapid changes in loop inductance. A microprocessor samples and analyzes the loop data and provides pulse or presence detection modes. Eight levels of sensitivity are also provided via a three position DIP switch on the front panel. The high setting ($S4 = 1$, $S2 = 1$, $S1 = 1$) was required for detection of logging trucks and other vehicles with areas of little metal mass (logging trucks often use the logs themselves for trailer linkage and support). Multiple loop frequencies, provided to prevent adjacent lane interference, are not relevant in this single lane application. Figure 30 illustrates the, active-low presence-mode, loop detector output signal (channel 1) with a corresponding weigh pad output signal (channel 2). Table 5 summarizes loop detector card Input File wiring.

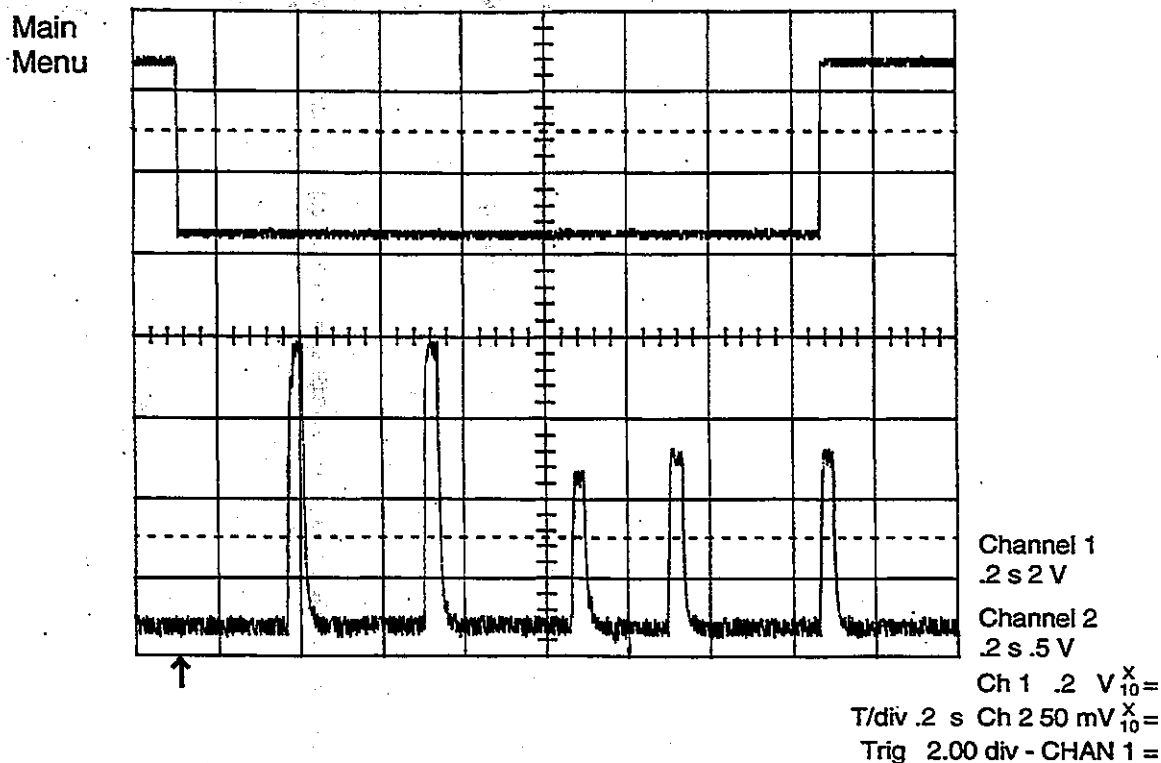


Fig. 30. DETECTOR SYSTEMS, INC., MODEL 222 LOOP DETECTOR OUTPUT SIGNAL (CHANNEL 1) AND WEIGH PAD OUTPUT SIGNAL (CHANNEL 2)

Terminal Strip # / Pin #	Function
3 / 1 - SP / H	Loop Out (E) *
2 - F	Loop Out (C) **
3 - W	
4 - D	Loop In
5 - E	Loop In
4 / 6 - J	+5V **
3 / 7 - K	
3 / 8 - L	Gnd *

NOTES:

* Emitter on Pin 3/1-SP/H tied to ground on Pin 3/8-L

** Collector on Pin 3/2-F pulled high through a 4.7 Kohm Resistor

Table 5. LOOP DETECTOR INPUT FILE WIRING

4.2.2.5 Miscellaneous

Weigh pad interface, axle sensor interface, and Peak Hold Detector cards are powered by the VME bus 12VDC supply. Power is supplied to the loop detector module with a standard 24VDC TSCES power supply. The differential amplifiers, VME bus, and 24 VDC TSCES power supply use the cabinet 120VAC which comes from the existing weigh station lighting circuit. Separate power supplies for these devices were used as a matter of convenience and, more importantly, to provide isolation and minimize interference. Nonetheless, supply problems were experienced. Transients, related to fluorescent lighting systems in the laboratory environment reset the ATC controller on several occasions. Two identical modular power supplies were tested in the controller in an attempt to isolate the problem. Both VME bus supplies were tested on clean stable circuits to verify nominal operation; both performed well.

In addition to miscellaneous controller hardware, a variety of material was required to complete the SWIM installation including: Fiberlite Pull Boxes, 1000 feet of #12 AWG loop conductor, circuit breakers, PVC and rigid conduit, elbows, couplers, and adapters, grounding rods, bushings, paint, lumber, epoxy, sand, concrete, and 4 1/2 tons of 1/2 inch asphalt. Complete installation details are provided in Appendix C.

4.2.3 Cost

SWIM system costs included a variety of devices and materials. The ATC prototype field controller, described previously, cost \$6280 (\$6100 hardware and \$180 software). The weighing system, including weigh pad sensors and frames (2) at \$8000 each, weigh pad interface card (1) at \$1500, epoxy (7 buckets) at \$100 each, and installation supervision at \$1500, came to a total of \$19,700. The axle sensor system, consisting of 12 Dynax sensors, frames, and epoxy at \$ 2794 each and 3 interface cards at \$147 each, came to a total of \$17,469. Miscellaneous material such as pvc conduit, conductor, concrete, asphalt, paint, and lumber totaled \$1394. Contract services,

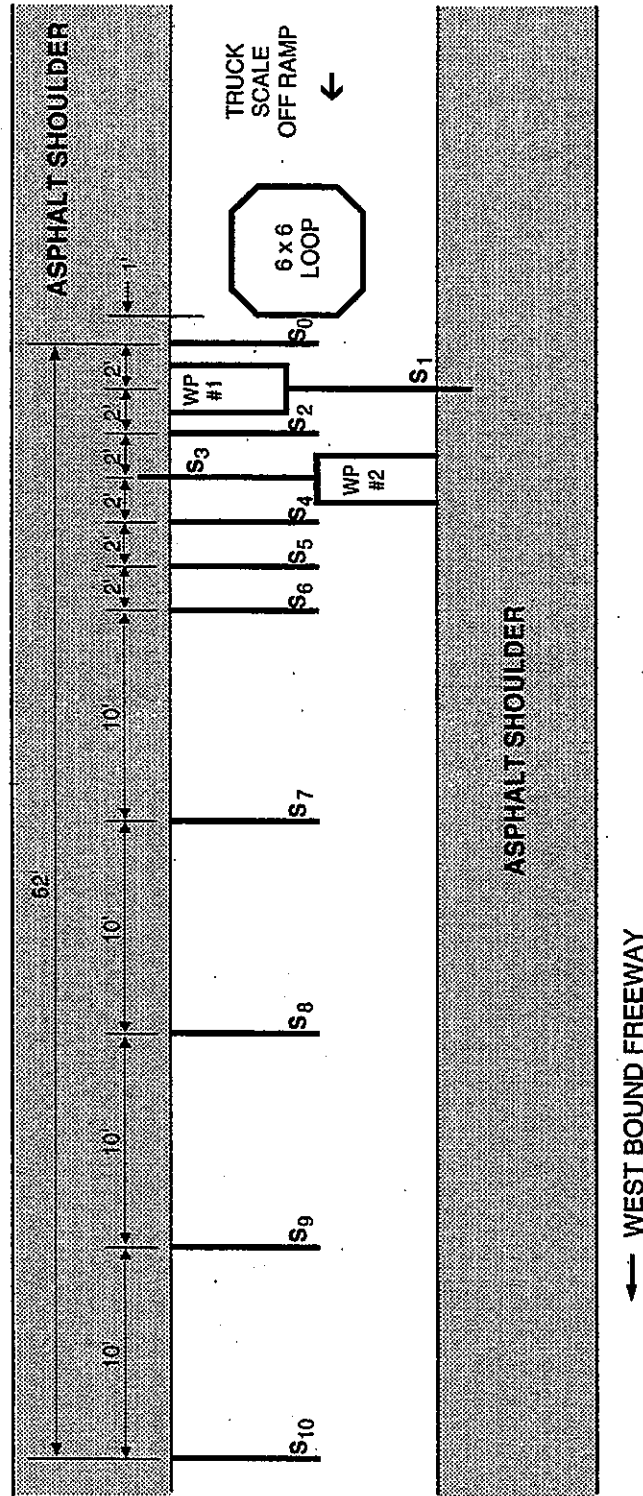
back hoe and pavement cutting, totaled \$1931. All items combined came to a grand total of \$46,774.

4.2.4 Operational Test

As stated previously, ATC prototype field testing targeted a control application requiring more processing power than current controller standards could provide. The SWIM system was a process intensive application requiring the use of a real-time operating system, high-level language, and a variety digital and analog I/O. In addition, the field test presented harsh physical conditions for environmental testing of the VME bus hardware.

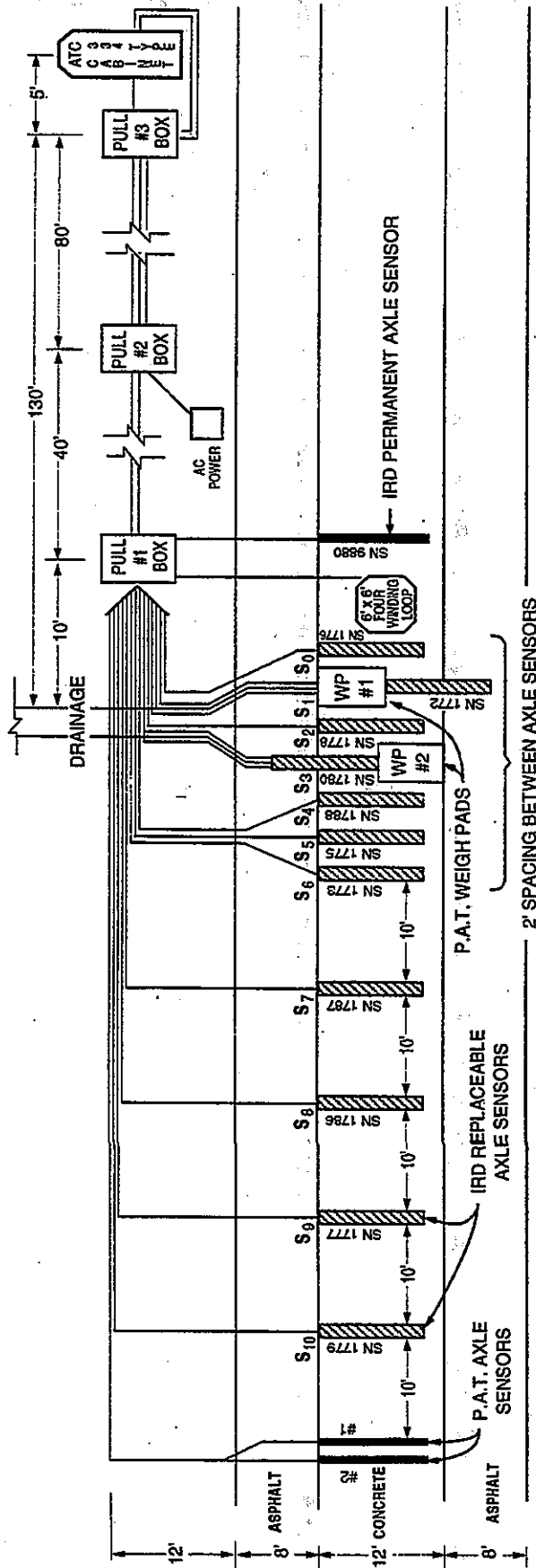
4.2.4.1 Functional Description

Numerous SWIM configurations were considered in the ATC prototype testing. The final design, illustrated in Figure 31, takes into account roadway geometry, fluctuating vehicle speeds, and provides two methods to determine peak weight amplitudes (via peak hold detector circuitry and/or sensor 1 and 3 hits). When the loop senses the presence of a vehicle, a new truck record is open. Axle sensors spaced 2 feet apart are used to determine accurate speeds and axle sensors spaced 10 feet apart are used to determine axle spacings. Weights are collected as described in the peak hold detector section. Figure 32 shows the actual ATC/SWIM prototype installation. The IRD permanent axle sensor and the PAT permanent axle sensors at either end of the system are not part of the prototype design; they were installed for independent test purposes only.



I80 / ANTELOPE WEIGH STATION

Fig. 31. BASIC ATC SWIM SYSTEM CONFIGURATION



180 / ANTELOPE RD. WEIGH-IN-MOTION (WIM)

NOTES:

- 1) Axle sensor and loop runs in separate 3/4" conduit t pull box #1, combined in 2" conduit pull box #1 to #3.
- 2) Weigh pad runs in separate 1-1/2" rigid conduit to pull box #1, combined in 1-1/2" rigid pull box #1 to #3.
- 3) Weigh pad drainage in 1-1/2" rigid.
- 4) Two spare conductors from pull box #1 to pull box #3, spare #1 → X1 and spare #2 → X2
- 5) P.A.T. axle sensors terminate at pull box #1 and spliced at asphalt edge as follows:
 SENSOR #1: white-red, green-gray
 SENSOR #2: black-red, red-gray
- 6) AC power spliced from CMS circuit (breaker #1 in scale house).
- 7) Weigh pad conductors spliced at pull box #1 as follows:
 WEIGHT PAD #1:
 white - white / green
 green - blue / white
 yellow - white / blue
 brown - green / white
 WEIGHT PAD #2:
 white - brown / white
 green - orange / white
 yellow - white / orange
 brown - white / brown
- 8) See attached documents for cabinet wiring diagrams.
- 9) See attached documents for weigh pad and axle sensor saw cut details.

Fig. 32. ATC SWIM PROTOTYPE SYSTEM INSTALLATION

4.2.4.2 Software

Several routines service axle sensors and weigh pads that are collecting axle and wheel load data. A "compute" routine processes acquired data, producing axle counts, axle loads, axle group loads, gross vehicle weights, speeds, center-to-center axle spacings, axle group spacings, vehicle classes (one of fifteen based on axle spacings and weights), site identifications, date and time stamps, sequential vehicle record numbers, California Vehicle Code (CVC) weight violations (including Bridge Law violations), and a display of summary data.

Figure 33, illustrates the SWIM operational concept. Numerous functions within the IRQ_DIN Module service sensors and initialize, open/close, activate/deactivate, and login/logout devices. Flow diagrams, Figures 34 through 45, detail sensor service routines. Appendix D provides complete source code listings.

In general, timers are triggered when each of the two foot spaced axle sensors are hit. These sensors, 0 through 5, are referred to as Speed Sensors (SS). Timers provide relative times (t_r) between Speed Sensor hits. Axle sensors 6 through 10 are referred to as Reference Sensors (RS). When a reference sensor is hit, it looks back to the last speed sensor hit and stops that sensor's timer. This provides the elapsed time between the reference sensor hit and the speed sensor hit (t_o). From t_r vehicle speed, relative to a specific axle, can be determined ($V = (2 \text{ foot spacing})/t_r$). The distance the axle traveled since the reference sensor was hit can be determined using the speed and the time elapsed between the reference sensor hit and the speed sensor hit ($D = V * t_o$). Axle spacing can now be determined from the known sensor positions and the distance the axle traveled since the reference sensor was hit ($AS = RS \text{ Position} - SS \text{ Position} - D$). This method provides axle spacings as a function of axle speeds, which, for slow-speed WIM, typically vary from axle to axle.

Once a complete set of data has been collected for a particular axle, a file consisting of the RS number, SS number, t_T time, and t_O time is transferred to the compute routine. For example, for the file [7,4,0.0299,0.0358], where $RF = 7$, $SS = 4$, $t_T = 0.0299$, and $t_O = 0.0358$, $V = 2/0.0299 = 66.89$ ft/sec, $D = 66.89 * 0.0358 = 2.39$ ft, and $AS = 22 - 8 - 2.39 = 11.61$ ft. When the loop signal drops out, indicating the absence of a vehicle, axle sensor 0's count is recorded as "Total Axle Count". As the count of each subsequent axle sensor reaches the "Total Axle Count", it is reinitialized. This process allows simultaneous processing of multiple vehicles in the SWIM detection zone. Figure 46 illustrates the ATC SWIM system truck record display screen.

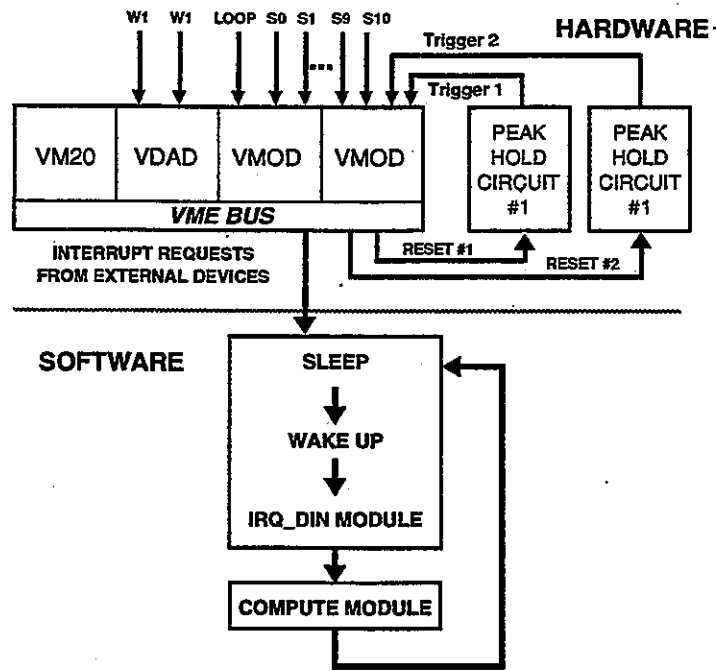


Fig. 33. ATC SWIM PROTOTYPE SYSTEM OPERATIONAL CONCEPT

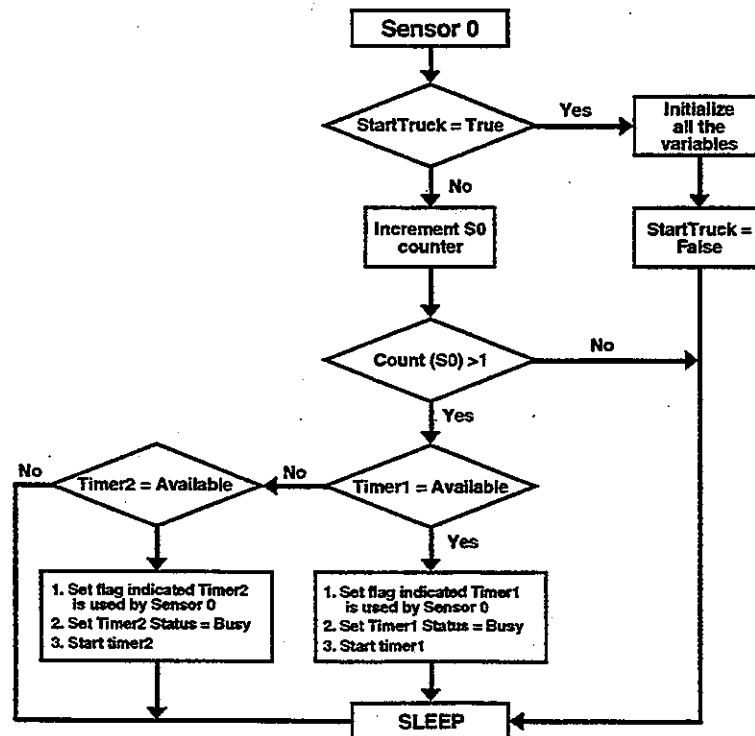


Fig. 34. ATC PROTOTYPE SWIM SOFTWARE SENSOR 0 SERVICE ROUTINE

Sensor 1 ->5 Service Routine

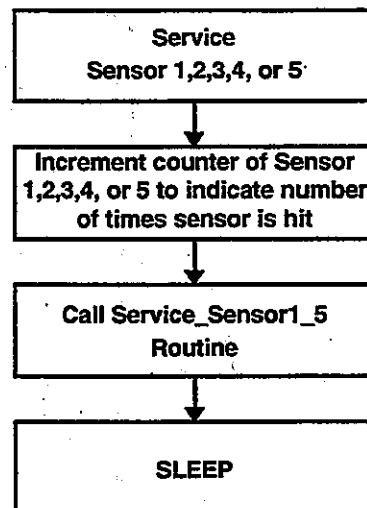


Fig. 35. ATC PROTOTYPE SWIM SOFTWARE
SENSOR 1-5 SERVICE ROUTINE

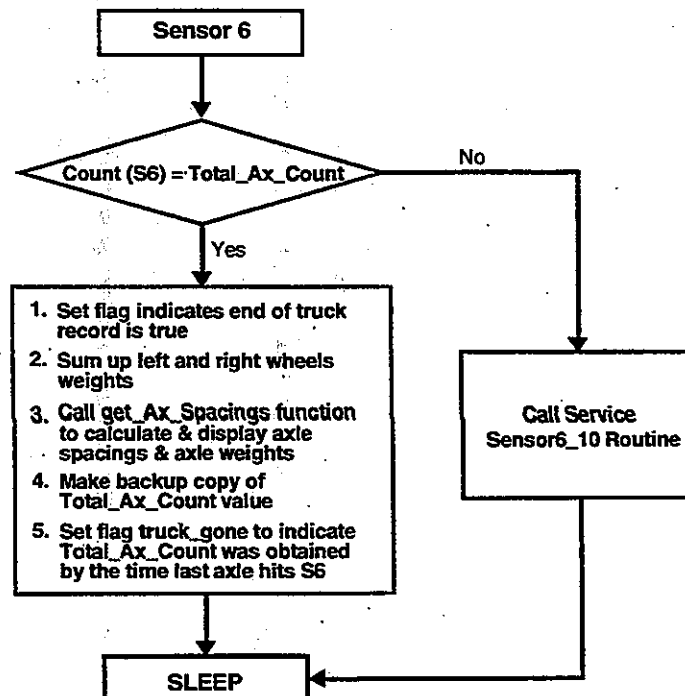


Fig. 36. ATC PROTOTYPE SWIM SOFTWARE
SERVICE SENSOR 6 ROUTINE

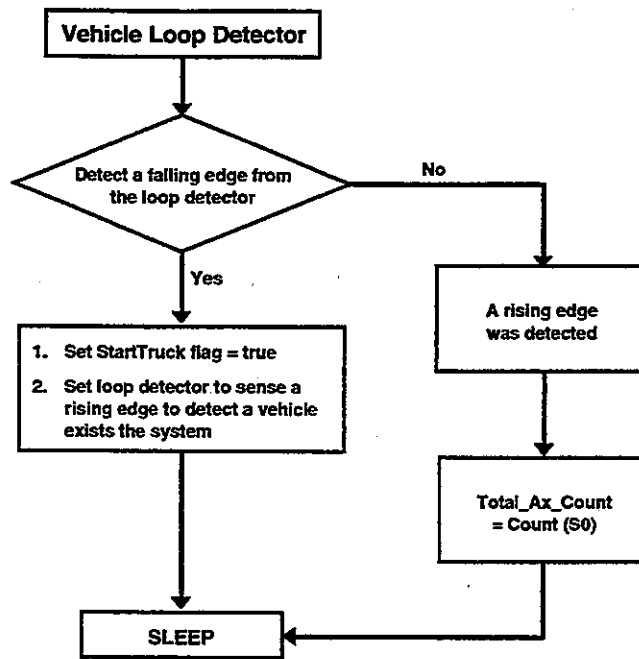


Fig. 39. ATC PROTOTYPE SWIM SOFTWARE VEHICLE LOOP DETECTOR SERVICE ROUTINE

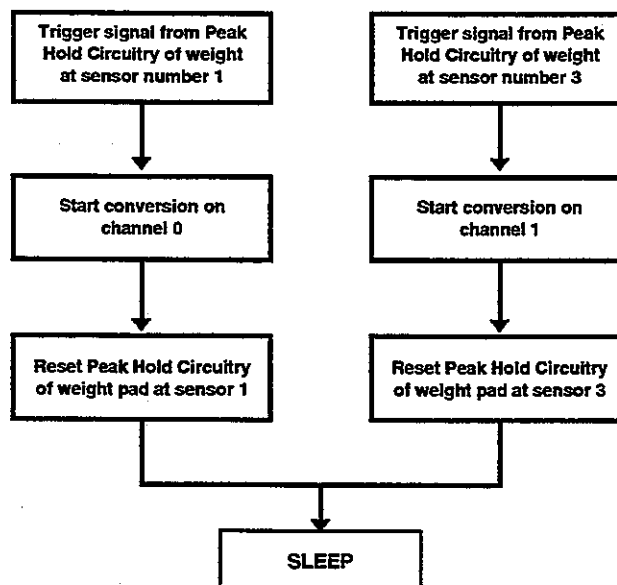


Fig. 40. ATC PROTOTYPE SWIM SOFTWARE VDAD ROUTINE

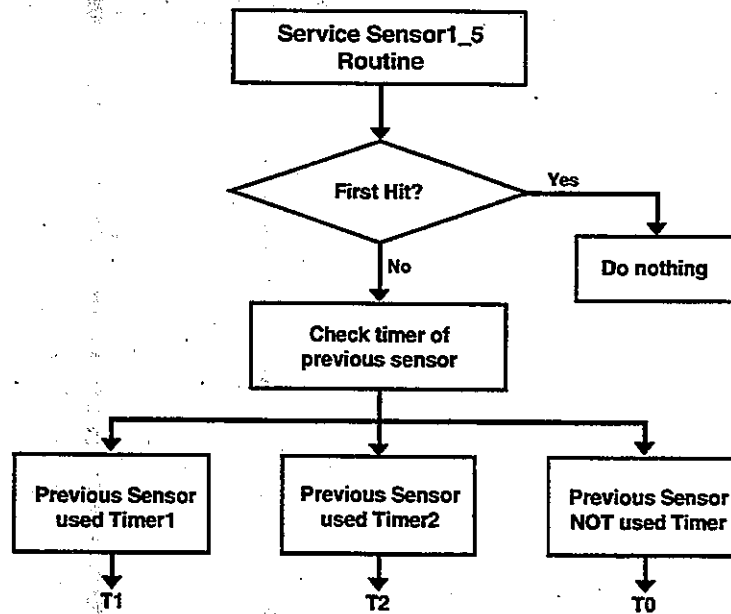


Fig. 41. ATC PROTOTYPE SWIM SOFTWARE SERVICE SENSOR 1 - 5 ROUTINE

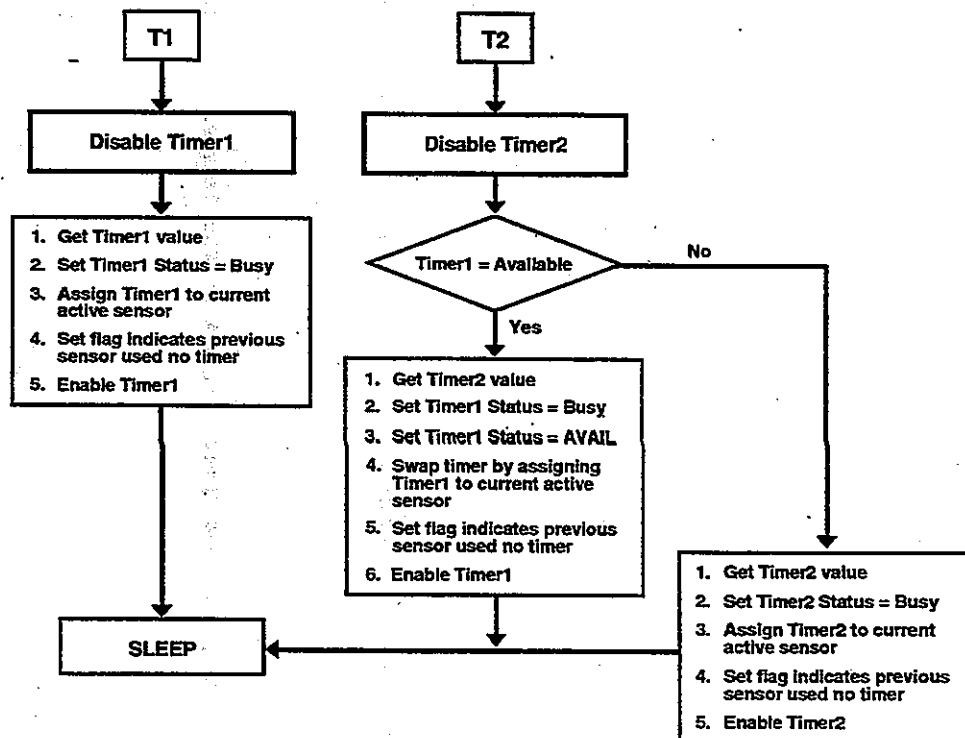


Fig. 42. ATC PROTOTYPE SWIM SOFTWARE SERVICE SENSOR 1 - 5 ROUTINE CONTINUED

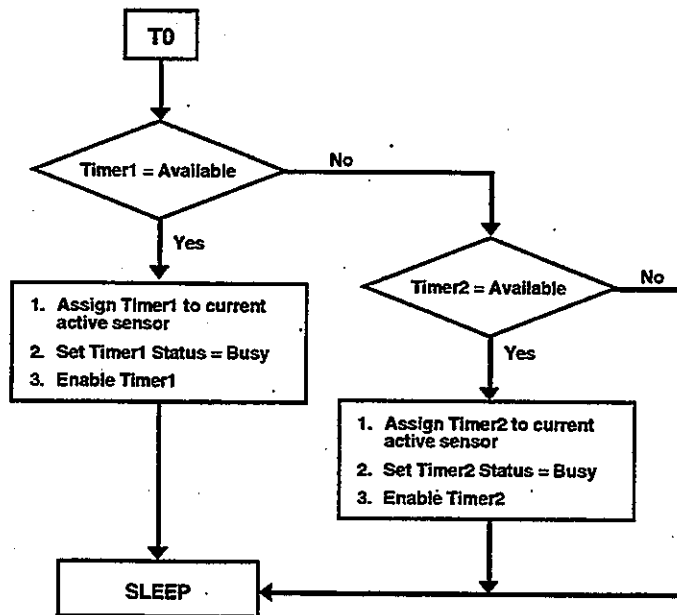


Fig. 43. **ATC PROTOTYPE SWIM SOFTWARE SERVICE SENSOR 1 - 5 ROUTINE CONTINUED**

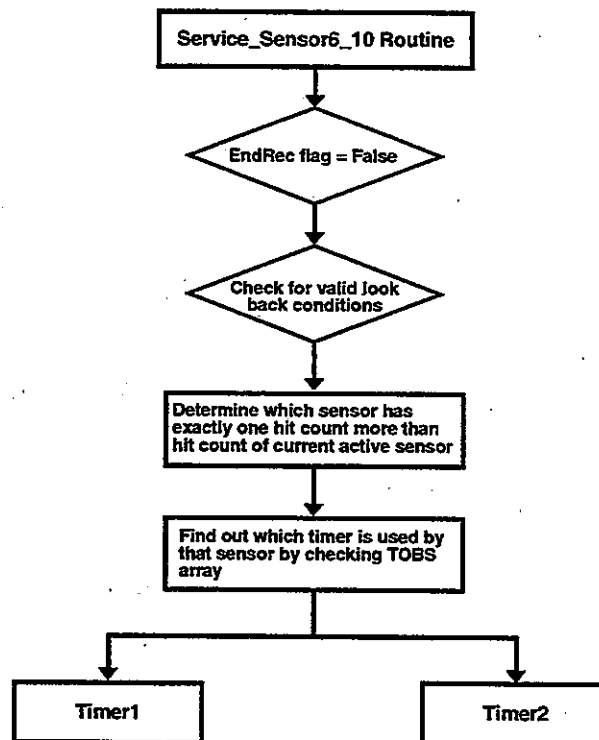


Fig. 44. **ATC PROTOTYPE SWIM SOFTWARE SERVICE SENSOR 6 - 10 ROUTINE**

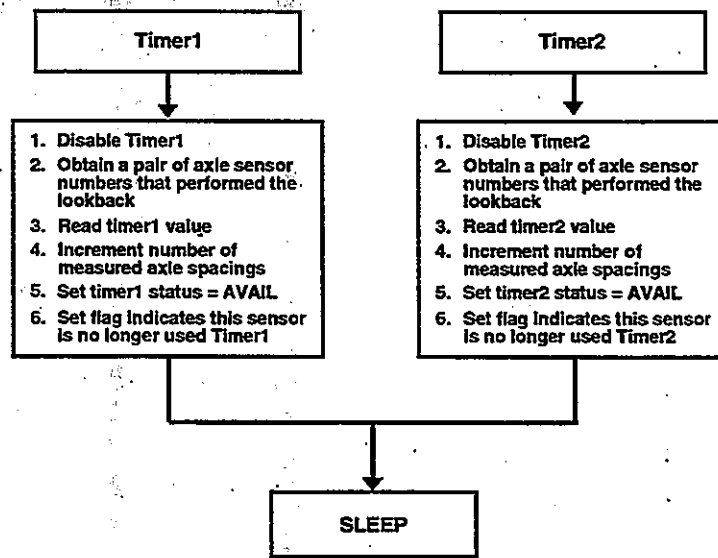


Fig. 45. ATC PROTOTYPE SWIM SOFTWARE SERVICE SENSOR 6 - 10 ROUTINE CONTINUED

Site #: Antelope Rd.				Date / Time: Tue Oct. 27 / 22:21:29 / 1992					
Vehicle Number: 1		Class: 13		Average Speed: 38.05 mph					
Axle:	1	2	3	4	5	6	7	8	9
Weight:	14400	25000	14000	19000	21000	10000	25000		
Spacing:	15.09	19.89	5.37	15.09	19.89	5.37			
VIOLATION SUMMARY									
Steering Axle >12500: 14400									
Single Axle >20000									
(axle / weight): 2 / 25000 5 / 21000 7 / 25000									
Tandem Axle >34000									
(Tandem / weight): 6-7 / 35000									
Gross Weight >80000: 128400									
Bridge Law Violations: 10									

Fig. 46. ATC SWIM SYSTEM TRUCK RECORD DISPLAY SCREEN

4.2.4.3 Evaluation

The prototype ATC SWIM system was evaluated for its overall performance, accuracy, functional adequacy, and physical reliability. Weight accuracy was based on a comparison of SWIM weights collected by the ATC system and static weights obtained from the static weigh scales at the Antelope weigh station. Axle spacing accuracy was determined by comparing SWIM data with measurements taken manually from stopped vehicles. Speed was not directly evaluated, as axle spacings are a function of speed and will reflect speed inaccuracies. Two sets of data were collected: one random set based on a single reading per vehicle for a number of different vehicles selected in random order from the traffic stream, and one controlled set based on a number of readings for a single calibrated vehicle.

The system was calibrated both statically and dynamically for weight and axle spacing measurements. Very little static calibration was required, as static SWIM weights measured very closely to known static weights. For dynamic calibration, a vehicle with known axle weights and spacings was driven through the system at a variety of speeds. A software calibration factor was set to match dynamic readings with known static parameters. Current California Department of Transportation and American Society for Testing and Materials WIM specifications require accuracy's listed in Tables 6 and 7; both tables refer to WIM systems providing over-weight screening in truck weigh stations. Wheel load refers to the sum of tire loads on all tires included in the wheel assembly which comprises a half axle (5). Single axle weight refers to the sum of all tire loads on a common mechanical axis oriented transversely to the direction of motion (5). Tandem axle weight refers to the sum of axle weights for any two consecutive axles with an axle spacing not exceeding 8.4 feet (6). Gross Weight refers to the total mass of the vehicle or the vehicle combination including all connected components (5). Axle spacing refers to the center-to-center length measurement between two axles, a consecutive pair or the first and last axles in a consecutive multi-axle group. Total wheel base refers to the length measurement between the first and last axles of the vehicle or the vehicle combination including all connected components (5, 6).

Table 7 requires the weight of a truck measured by a WIM system to be within $\pm 6\%$ of the weight of the truck measured by a static system, for 95% of the vehicles weighed. However, previous studies have shown that typical weight error data are not normally distributed (7) and should be referenced in terms of mean error and standard deviation as shown in Table 6. This mean error/standard deviation convention is adopted for ATC SWIM test.

FHWA vehicle types are described in Table 8 and summarized in Table 9. Non-passenger vehicles make up the majority of classes and are subdivided into several categories. Fifteen different classes exist and are used in Caltrans WIM operations and the ATC SWIM prototype system.

Parameter	Mean Error	Standard Deviation
Single Axle Weight	+/- 2%	3%
Tandem Axle Weight	+/- 2%	2%
Gross Weight	+/- 2%	1.5%
Axle Space	+/- 0.5'	0.5'
Total Wheel Base	+/- 1.0'	1.0'

Table 6. **CALIFORNIA DEPARTMENT OF TRANSPORTATION REQUIRED WIM SYSTEM ACCURACIES FOR SYSTEMS OPERATING WITH SPEEDS RANGING BETWEEN 3 AND 40 mph**

Tolerance for 95% Probability of Conformity	
Parameter	Error
Wheel Load	+/- 20%
Single Axle Weight	+/- 15%
Group Axle Weight	+/- 10%
Gross Weight	+/- 6%
Axle Space	+/- 0.5'
Speed	+/- 1.0 mph

Table 7. **AMERICAN SOCIETY FOR TESTING AND MATERIALS WIM SYSTEM ACCURACIES FOR SYSTEMS WITH OPERATING SPEEDS BETWEEN 15 AND 50 MPH**

Table 8. FWHA VEHICLE CLASSIFICATIONS WITH DEFINITIONS

- Class 1 - *Motorcycles:*** All two or three wheeled motorized vehicles including motorcycles, motor scooters, mopeds, motor-powered bicycles, and three-wheel motorcycles.
- Class 2 - *Passenger Cars:*** All sedans, coupes, station wagons, and mini-vans manufactured for the primary purpose of carrying passengers and includes passenger cars pulling recreational or other light trailers.
- Class 3 - *Other Two-Axle and Four-Axle Single Unit Vehicles:*** All two and four axle vehicles other than passenger cars, including pickups, panel vans, and other vehicles such as campers, motor homes, ambulances, hearses, and carryalls. Other two and four axle vehicles pulling recreation or other light trailers are included in this classification.
- Class 4 - *Buses:*** All vehicles manufactured as traditional passenger-carrying buses with two axles and six tires or three or more axles. This category includes only traditional buses (including school buses) functioning as passenger-carrying vehicles. All two and four axle mini-buses should be categorized as class three.
- Class 5 - *Two Axle, Six Tire, Single Unit Trucks:*** All vehicles on a single frame including trucks, camping and recreational vehicles, motor homes, etc., having two axles and dual rear wheels.
- Class 6 - *Three Axle Single Unit Trucks:*** All vehicles on a single frame including trucks, camping and recreational vehicles, motor homes, etc., having three axles.

Continues

Class 7 - Four or More Axle Single Unit Trucks: All trucks on a single frame with four or more axles.

Class 8 - Four or Less Axle Single Trailer Trucks: All vehicles with four or less axles consisting of two units, one of which is a tractor or straight truck power unit.

Class 9 - Five Axle Single Trailer Trucks: All 5 axle vehicles consisting of two units, one of which is a tractor or straight truck power unit.

Class 10 - Six or More Axle Single Trailer Trucks: All vehicles with six or more axles consisting of two units, one of which is a tractor or straight truck power unit.

Class 11 - Five or Less Axle Multi-Trailer Trucks: All vehicles with five or less axles consisting of three or more units, one of which is a tractor or straight truck power unit.

Class 12 - Six Axle Multi-Trailer Trucks: All six axle vehicles consisting of three or more units, one of which is a tractor or straight truck power unit.

Class 13 - Seven or More Axle Multi-Trailer Trucks: All vehicles with seven or more axles consisting of three or more units, one of which is a tractor or straight truck power unit.

Class 14 - Truck and Trailer: All vehicles with five axles consisting of two units, one being a straight truck power unit and the second a full trailer. Does not include semi's. See class nine for semi's.

Class 15 - All Other: All vehicles not classified by classes 1- 14.

Class	Vehicle Description	No. Axles	Axle Space 1-2	Axle Space 1-2	Axle Space 1-2	Axle Space 1-2	Axle Space 1-2	Axle Space 1-2	Axle Space 1-2	Axle Space 1-2	Axle Space 1-2	Gross Weight (Kips)	Notes
1	Motorcycle	2	00.1 - 06.0									00.1 - 03.0	
2	Auto, Pickup	2	06.1 - 09.9									01.0 - 06.0	
2	Auto w/ 1 Ax Trlr	3	06.1 - 09.9	06.1 - 18.0								01.0 - 09.9	
2	Auto w/ 2 Ax Trlr	4	06.1 - 09.9	06.1 - 18.0	00.1 - 03.4							01.0 - 09.9	
3	Other (Limo, Van, Rv, etc.)	2	10.0 - 13.3									01.0 - 09.9	
3	Other w/ 1 Ax Trlr	3	10.0 - 13.3	06.1 - 18.0								01.0 - 09.9	
3	Other w/ 12Ax Trlr	4	10.0 - 13.3	06.1 - 18.0	00.1 - 03.4							01.0 - 09.9	
3	Other w/ 3 Ax Trlr	5	10.0 - 13.3	06.1 - 18.0	00.1 - 03.4	06.1 - 44.0						01.0 - 09.9	
4	Bus	2	20.1 - 40.0									XXX - XXX	
4	Bus	3	20.1 - 40.0	03.5 - 06.0								XXX - XXX	
5	2D	2	13.4 - 20.0									00.0 - XXX	
6	3A	3	06.1 - 20.0	03.5 - 06.0								XXX - XXX	
7	4A	4	06.1 - 23.0	03.5 - 06.0	00.1 - 06.0							10.0 - XXX	
8	291, 21	3	09.5 - 23.0	18.0 - 40.0								10.0 - XXX	
8	391, 31	4	06.1 - 23.0	03.5 - 06.0	06.1 - 44.0							10.0 - XXX	
8	292	4	06.1 - 23.0	11.0 - 40.0	03.5 - 10.9							10.0 - XXX	
9	392	5	06.1 - 26.0	03.5 - 06.0	00.1 - 46.0	03.5 - 10.9						10.0 - XXX	
10	393, 33	6	06.1 - 26.0	03.5 - 06.0	00.1 - 46.0	00.1 - 11.0						10.0 - XXX	
11	2912	5	06.1 - 26.0	11.1 - 26.0	06.1 - 20.0	11.1 - 26.0						10.0 - XXX	
12	3912	6	06.1 - 26.0	03.5 - 6.0	11.1 - 26.0	06.1 - 20.0						10.0 - XXX	
13	2923, 3922, 3913	7	01.0 - 45.0	01.0 - 45.0	01.0 - 45.0	01.0 - 45.0	01.0 - 45.0	01.0 - 45.0	01.0 - 45.0	01.0 - 45.0		10.0 - XXX	
13	3923	8	01.0 - 45.0	01.0 - 45.0	01.0 - 45.0	01.0 - 45.0	01.0 - 45.0	01.0 - 45.0	01.0 - 45.0	01.0 - 45.0	01.0 - 45.0	10.0 - XXX	
13	Permit	9	01.0 - 45.0	01.0 - 45.0	01.0 - 45.0	01.0 - 45.0	01.0 - 45.0	01.0 - 45.0	01.0 - 45.0	01.0 - 45.0	01.0 - 45.0	10.0 - XXX	
14	32	5	06.1 - 23.0	03.5 - 06.0	06.1 - 23.0	11.0 - 27.0						10.0 - XXX	
15	Unclassified	Vehicles not meeting axle configurations set for Classifications 1 thru 14											

Table 9. FHWA VEHICLE WEIGHT / AXLE SPACING CLASSIFICATIONS

The controlled data set mentioned previously is based on a class 5 truck, a 2 axle vehicle with an axle spacing between 13.4 and 20 feet and weight over 8.0 kips (8000 lbs). Table 10 summarizes statistics related to data collected on this vehicle. The average percent error for axle spacing, axle 1 weight, axle 2 weight, and gross weight are listed for several speed groups. Figure 47 graphically illustrates the data presented in Table 10 with average percent error versus speed. Figure 48 shows average percent error versus axle spacing, axle weight, and gross weight for the same data set. The statistics in Table 11 are based on the same controlled data set but provide the standard deviation of the average percent error. These statistics are also graphically illustrated in Figures 49 and 50.

Test results show the average error for axle spacing is well within the accuracy criteria listed in Table 6. The test vehicle, with an axle spacing of 14.65 feet (equal to total wheel base spacing for a two axle vehicle), produced a 0.48% mean axle spacing error equating to 0.07 feet; far less than either the 1.0 foot total wheel base maximum or 0.5 foot axle spacing maximum. However, weight errors were found to be slightly higher, at 3.89% single axle error (axle 1 and axle 2 averaged) and 3.9% gross weight error, than the +/- 2% figures listed in Table 6. Weight errors were greatest in the 10 - 20 mph range as clearly illustrated in Figures 47 and 48. Although average weight errors were high, standard deviations were acceptable, with spacings within 0.25 feet measured vs. 0.5 feet required and single axle weights at 2.27% measured vs. 3% required (axle 1 and axle 2 standard deviations averaged). Gross weight standard deviation was slightly higher than the allowable 1.5% at 1.74%.

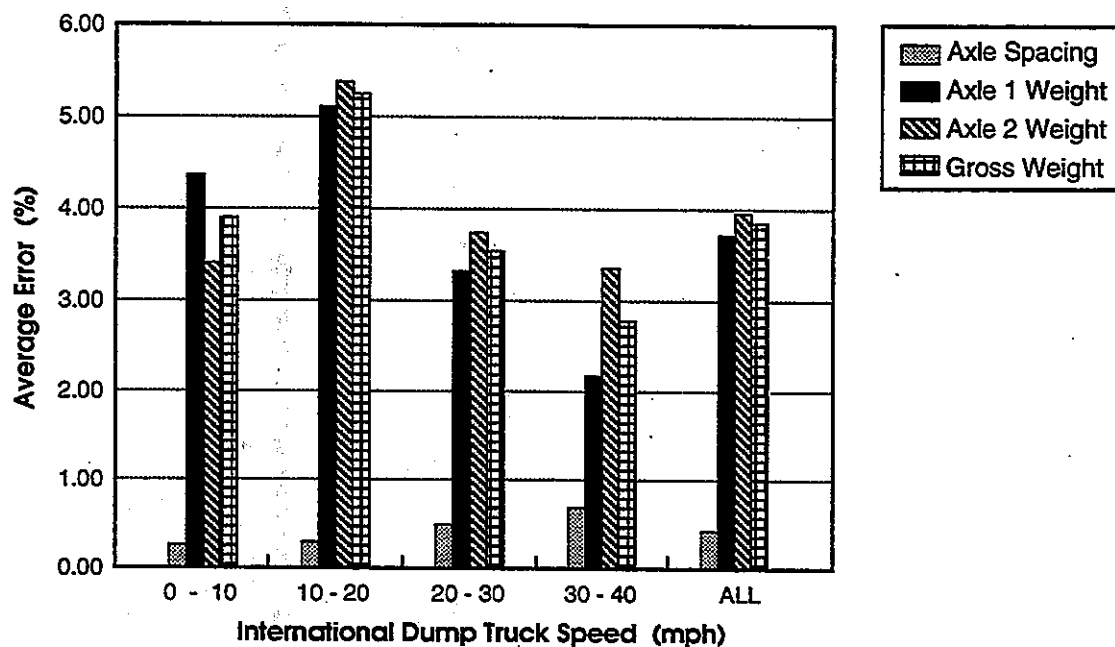


Fig. 47. **CONTROLLED DATA SET - AVERAGE PERCENT ERROR VERSUS SPEED**

Speed Groups	Avg, - mph	Spacing	Axle 1	Axle 2	Gross Weight
0 - 10	8.29	0.29	4.42	3.46	3.94
10 - 20	13.64	0.34	5.11	5.4	5.26
20 - 30	26.95	0.55	3.38	3.76	3.57
30 - 40	33.05	0.75	2.22	3.41	2.82
ALL	20.48	0.48	3.78	4	3.9

Table 10. **CONTROLLED DATA SET AVERAGE ERROR**

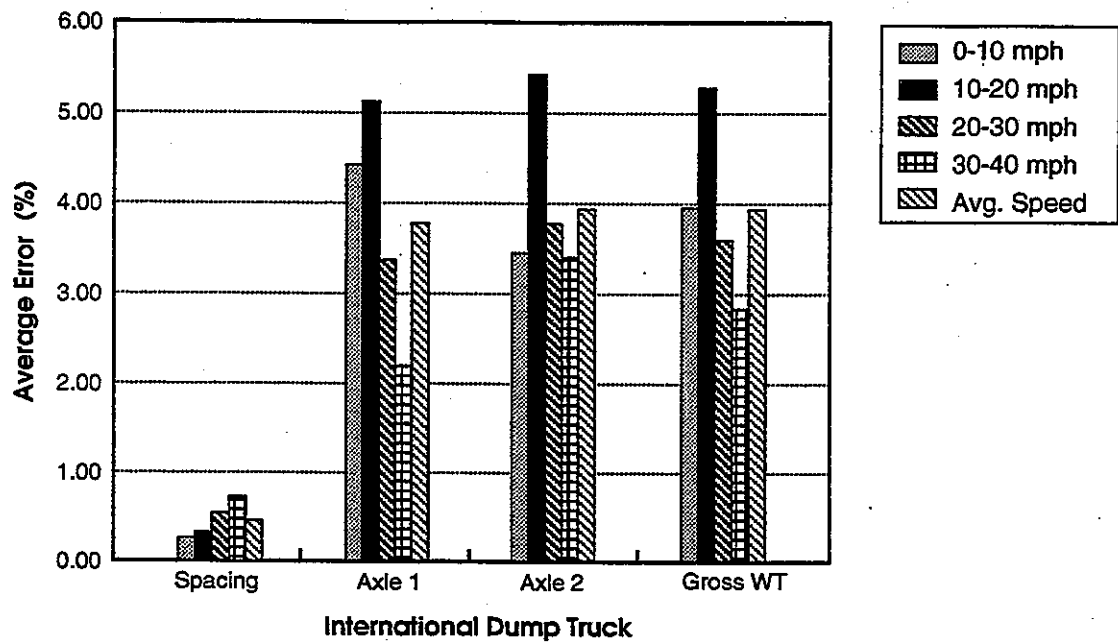


Fig. 48. **CONTROLLED DATA SET - AVERAGE PERCENT ERROR VERSUS AXLE SPACING, AXLE WEIGHT, AND GROSS WEIGHT**

Speed Groups	Avg, - mph	Spacing	Axle 1	Axle 2	Gross Weight
0 - 10	8.29	0.1	2.44	1.18	1.8
10 - 20	13.64	0.25	1.2	2.29	1.71
20 - 30	26.95	0.08	2.14	3.35	1.52
30 - 40	33.05	0.16	1.48	1.61	1.1
ALL	20.48	0.25	2.16	2.38	1.74

Table 11. **CONTROLLED DATA SET STANDARD DEVIATION OF AVERAGE ERROR**

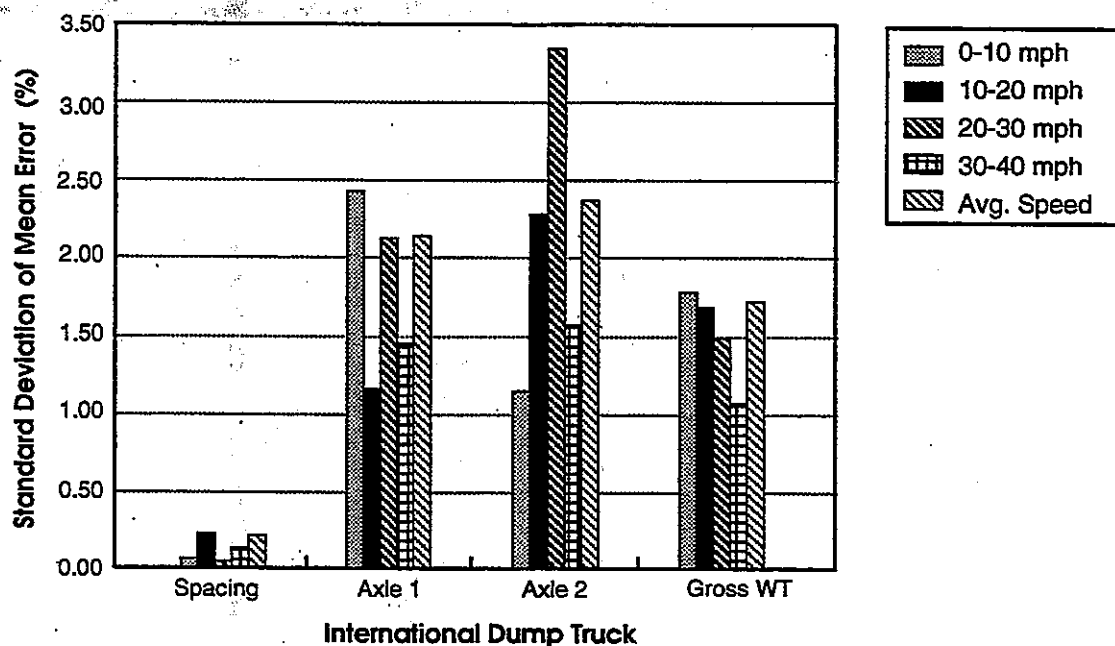


Fig. 49. CONTROLLED DATA SET - STANDARD DEVIATION OF MEAN ERROR VERSUS AXLE SPACING, AXLE WEIGHT, AND GROSS WEIGHT

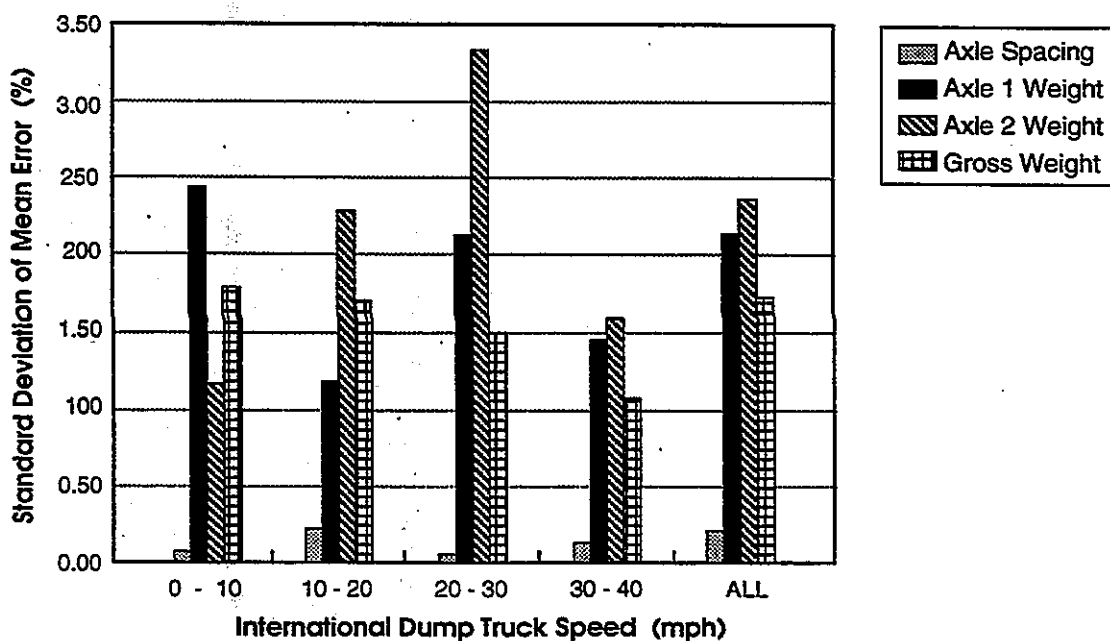


Fig. 50. CONTROLLED DATA SET - STANDARD DEVIATION OF MEAN ERROR VERSUS SPEED

The second set of test data, as described previously, was derived from a group of random vehicles. Resulting statistics are listed in Table 12 and Table 13, average percent error and standard deviation of average percent error respectively. Here, gross weight, over all length (OAL), single axle weight, and axle spacing average percent errors are listed for several classifications. Figure 51 illustrates average percent error vs. class. Figure 52 illustrates average percent error vs. gross weight, OAL, single axle weight, and axle spacing. Figure 53 illustrates standard deviation of average percent error vs. class. Figure 54 illustrates standard deviation of average percent error vs. gross weight, OAL, single axle weight, and axle spacing. The same data set was used to develop Tables 14 and 15, average percent error and standard deviation of average percent error respectively, with respect to speed, rather than class as in Tables 12 and 13. Corresponding graphs are provided in Figures 55 through 58.

The random data again shows spacing accuracies well within allowable limits. Based on axle spacing and OAL errors of 1.18% and 0.76% respectively, a worst case scenario, 75 foot double rig and 40 foot trailer, would result in 0.47 foot axle spacing error and 0.57 foot over all length error. Both figures are below required limits of 0.5 foot axle spacing error and 1.0 foot total wheel base error. However, weight errors are again higher than allowable limits. Gross weight resulted in a 3.39% average error and single axle weights were found to be 4.07%. Both figures are greater than the specified $\pm 2\%$. Standard deviations for over all axle spacing and single axle weight were within the 1.0 foot limit at 0.65 feet and 3.0% limit at 2.53% respectively. Other standard deviations were above acceptable limits as follows: axle spacing measured 0.69 feet vs. 0.5 feet required and gross weight measured 2.2% vs. 1.5% required. Figures 51 through 58 show a definite trend toward greater inaccuracies for speeds between 10 and 20 mph as well as a significant increase in error for classes 11 and 14.

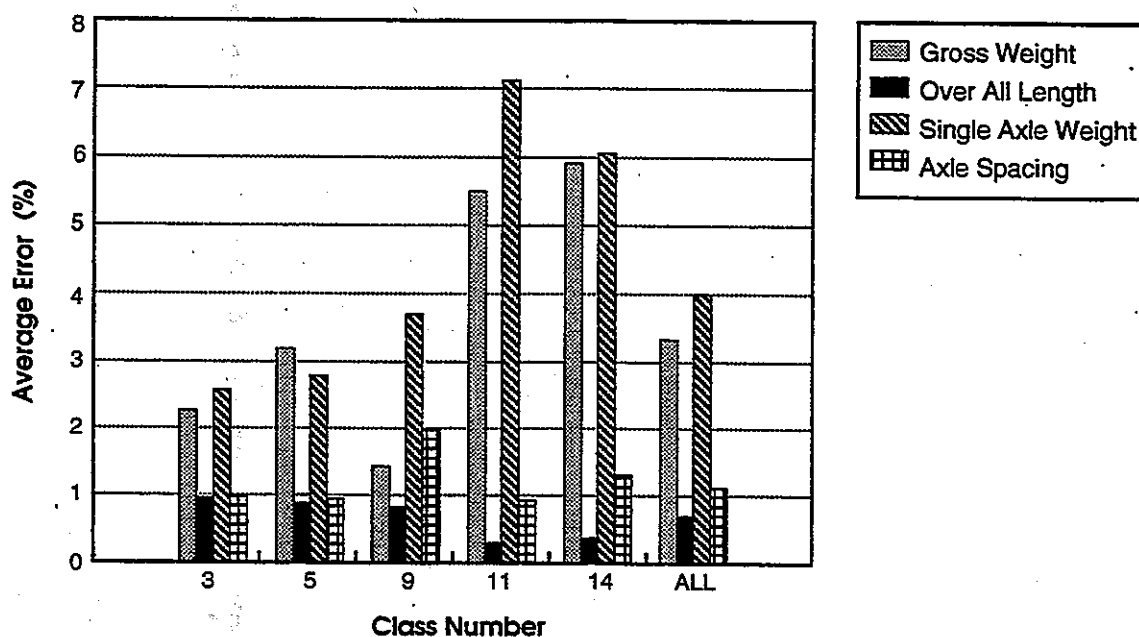


Fig. 51. **RANDOM DATA SET - AVERAGE PERCENT ERROR - VERSUS CLASS**

Class	Gross Weight	OAL	Single Axle	Axle Space
3	2.3	1.03	2.61	1.03
5	3.22	0.94	2.83	1.03
9	1.48	0.9	3.75	2.06
11	5.57	0.23	7.15	0.99
14	5.97	0.28	6.11	1.38
ALL	3.39	0.76	4.07	1.18

Table 12. **RANDOM DATA SET AVERAGE ERROR WITH. RESPECT TO CLASS**

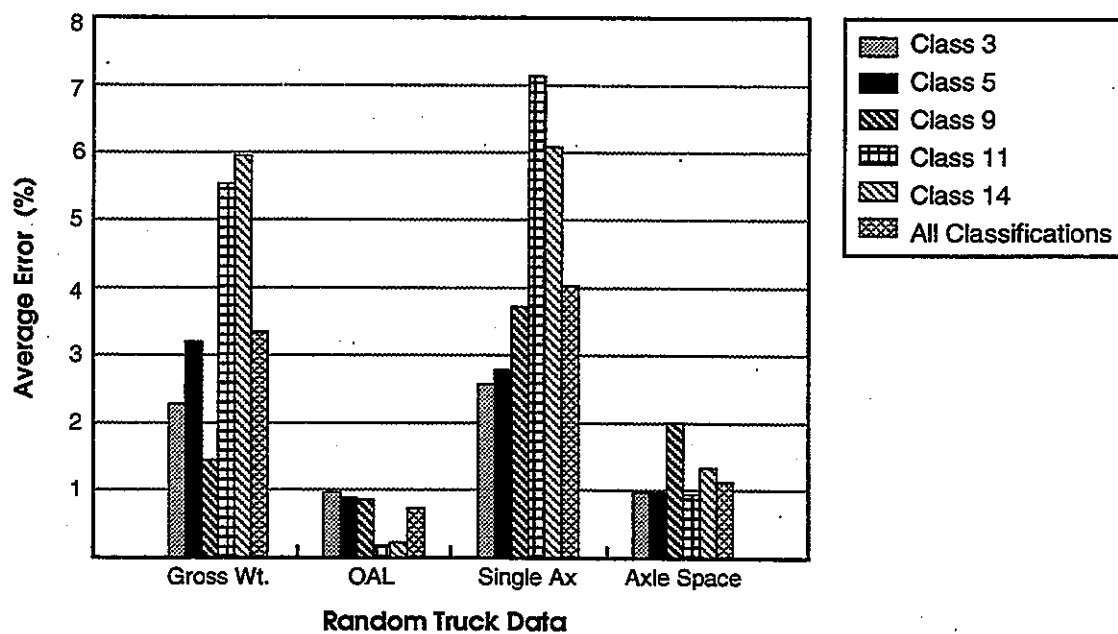


Fig. 52. **RANDOM DATA SET - AVERAGE PERCENT ERROR VERSUS AXLE SPACING, AXLE WEIGHT, GROSS WEIGHT, AND OVER-ALL LENGTH WITH RESPECT TO CLASS**

Class	Gross Weight	OAL	Single Axle	Axle Space
3	0.76	0.82	1.25	0.82
5	2.38	0.02	1.73	0.02
9	0.29	0.22	2.14	0.41
11	2.63	0.2	2.5	0.46
14	1.09	0.01	1.06	0.09
ALL	2.26	0.65	2.53	0.69

Table 13. **RANDOM DATA SET STANDARD DEVIATION OF AVERAGE ERROR WITH RESPECT TO CLASS**

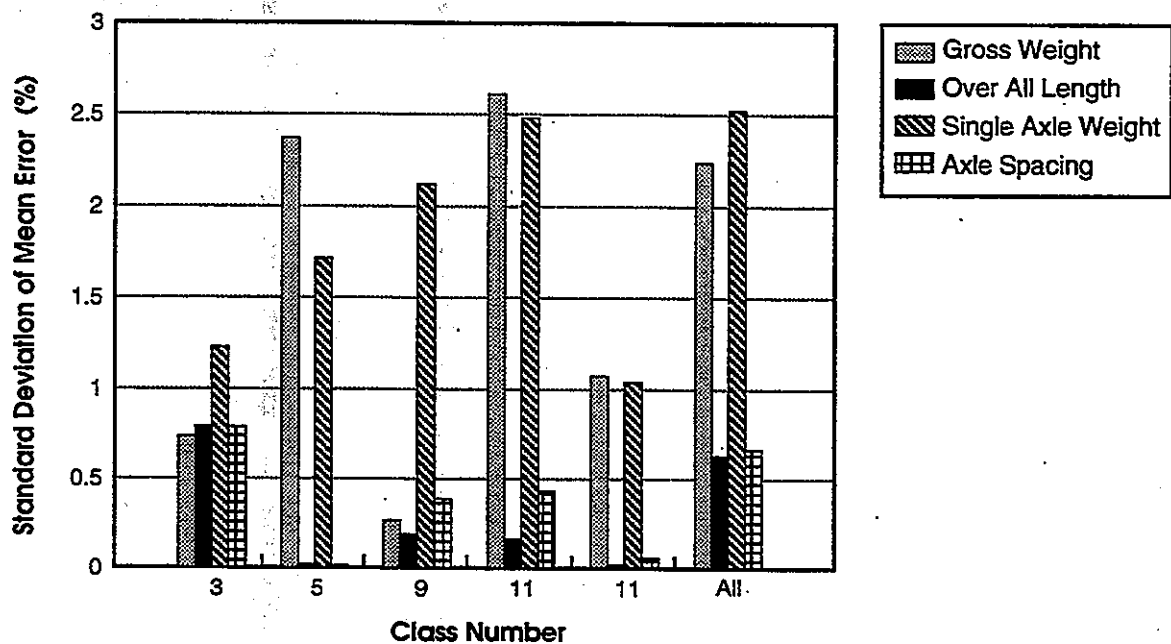


Fig. 53. RANDOM DATA SET - STANDARD DEVIATION OF MEAN ERROR VERSUS CLASS

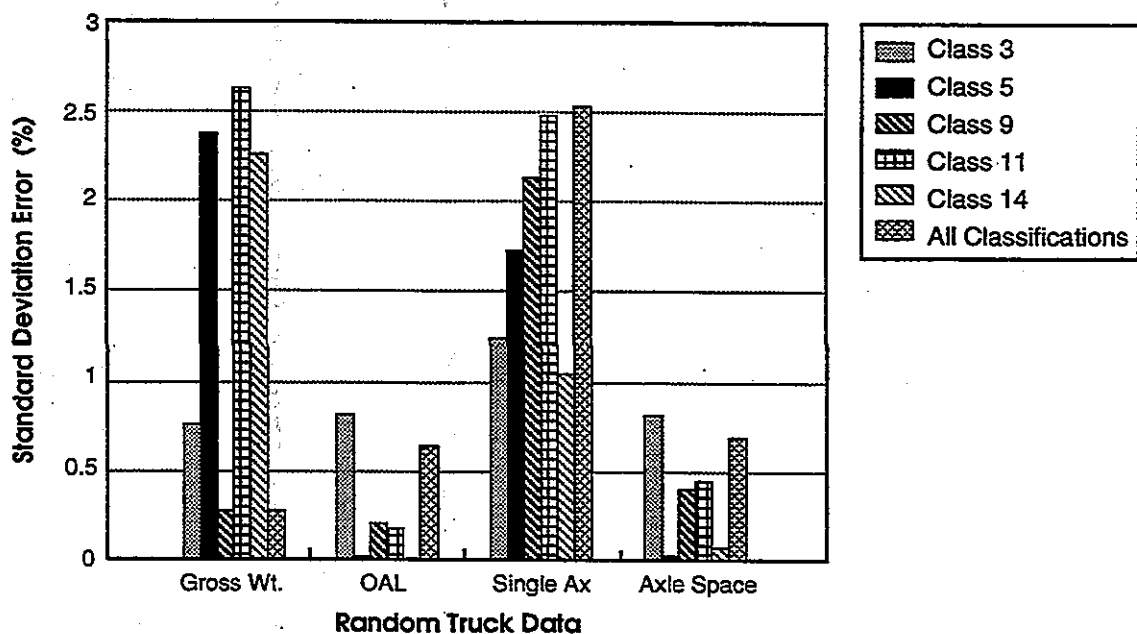


Fig. 54. RANDOM DATA SET - STANDARD DEVIATION OF MEAN ERROR VERSUS AXLE SPACING, AXLE WEIGHT, GROSS WEIGHT, AND OVER-ALL LENGTH WITH RESPECT TO CLASS

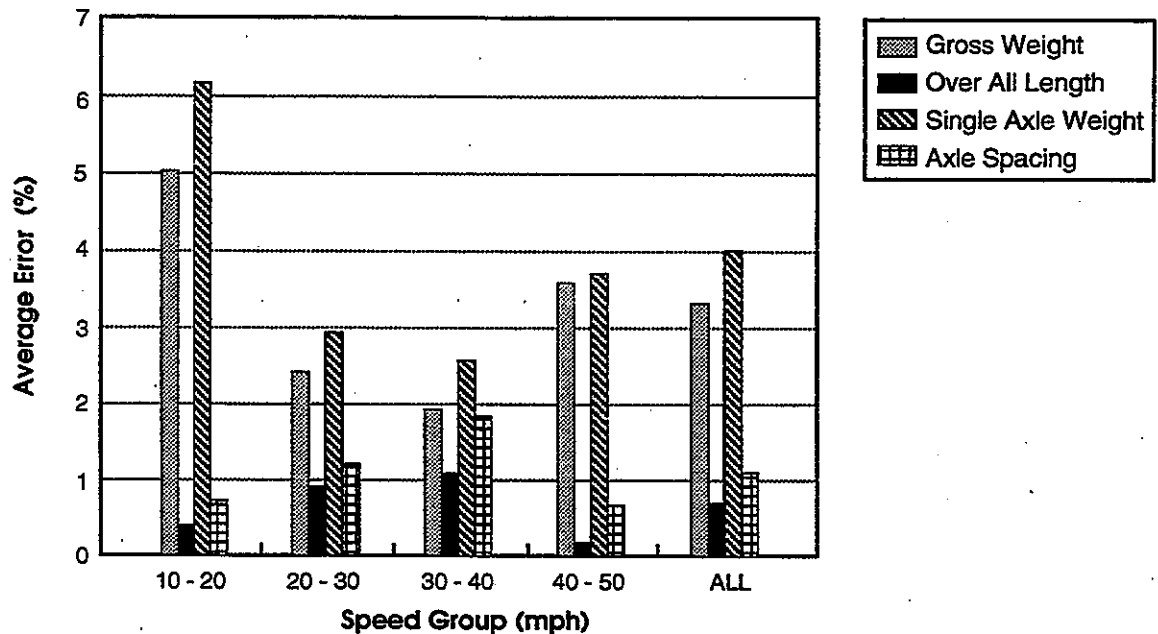


Fig. 55. **RANDOM DATA SET - AVERAGE PERCENT ERROR VERSUS SPEED**

Speed	Gross Weight	OAL	Single Axle	Axle Spacing
10 - 20	5.09	0.48	6.23	0.831
20 - 30	2.49	0.99	3.01	1.32
30 - 40	2.01	1.16	2.66	1.91
40 - 50	3.65	0.23	3.77	0.75
ALL	3.39	0.76	4.07	1.18

Table 14. **RANDOM DATA SET AVERAGE ERROR WITH RESPECT TO SPEED**

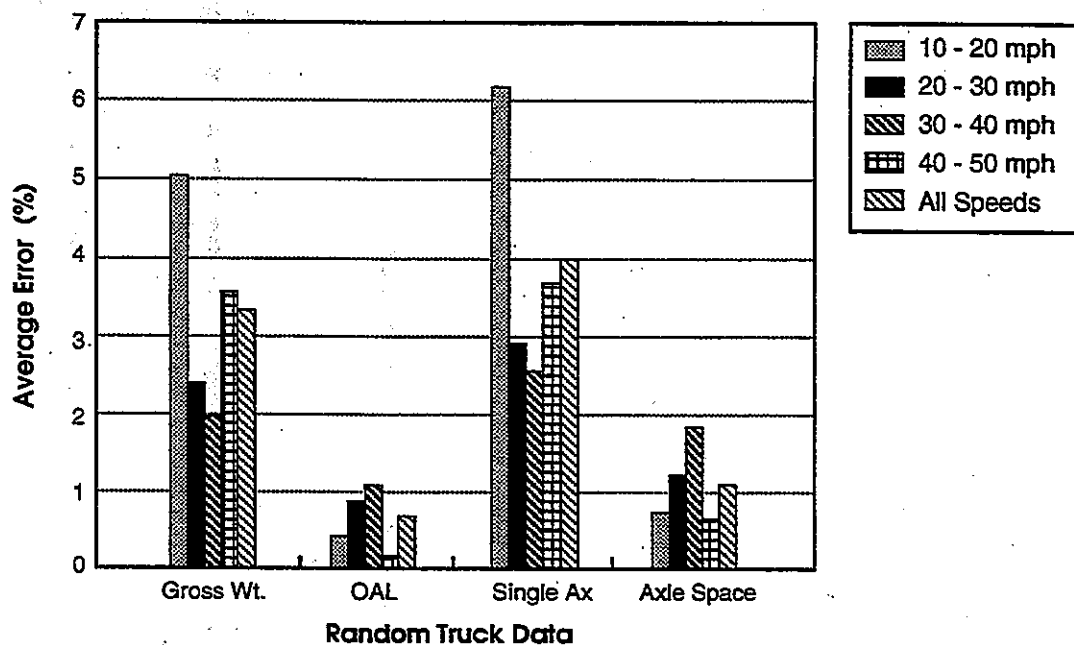


Fig. 56. **RANDOM DATA SET - AVERAGE PERCENT ERROR VERSUS AXLE SPACING, AXLE WEIGHT, GROSS WEIGHT, AND OVER-ALL LENGTH WITH RESPECT TO CLASS WITH RESPECT TO SPEED**

Speed	Gross Weight	OAL	Single Axle	Axle Spacing
10 - 20	2.82	0.26	2.95	0.39
20 - 30	1.45	0.56	1.58	0.75
30 - 40	0.46	1.14	1.13	0.39
40 - 50	1.23	0.04	1.28	0.55
ALL	2.26	0.65	2.53	0.69

Table 15. **RANDOM DATA SET DEVIATION OF AVERAGE**

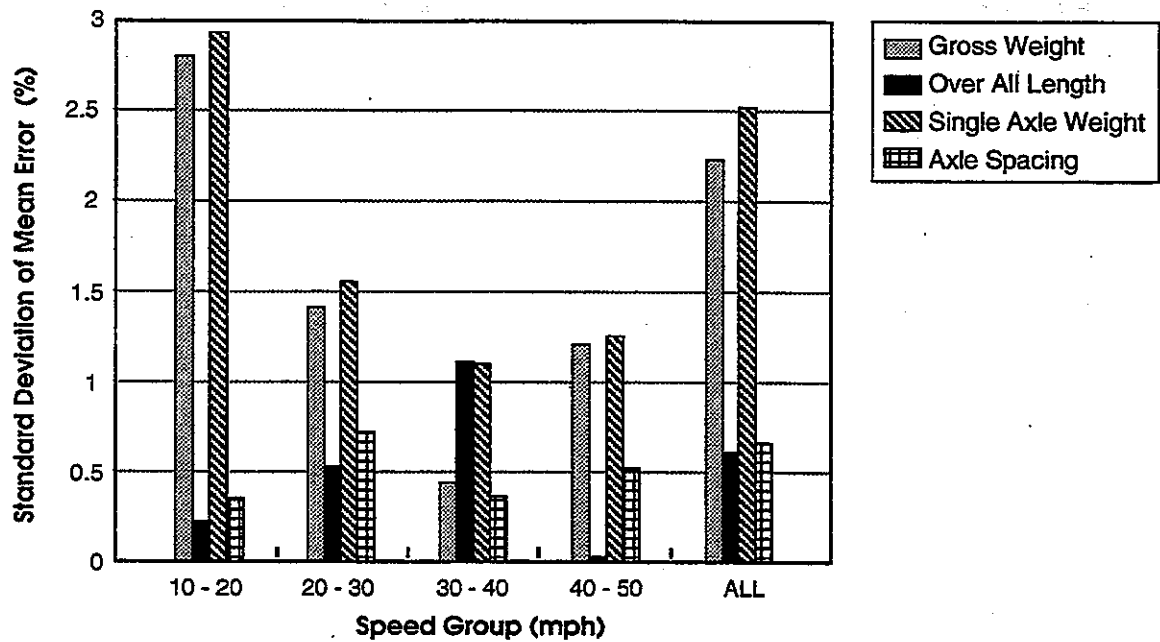


Fig. 57. **RANDOM DATA SET - STANDARD DEVIATION OF MEAN ERROR**

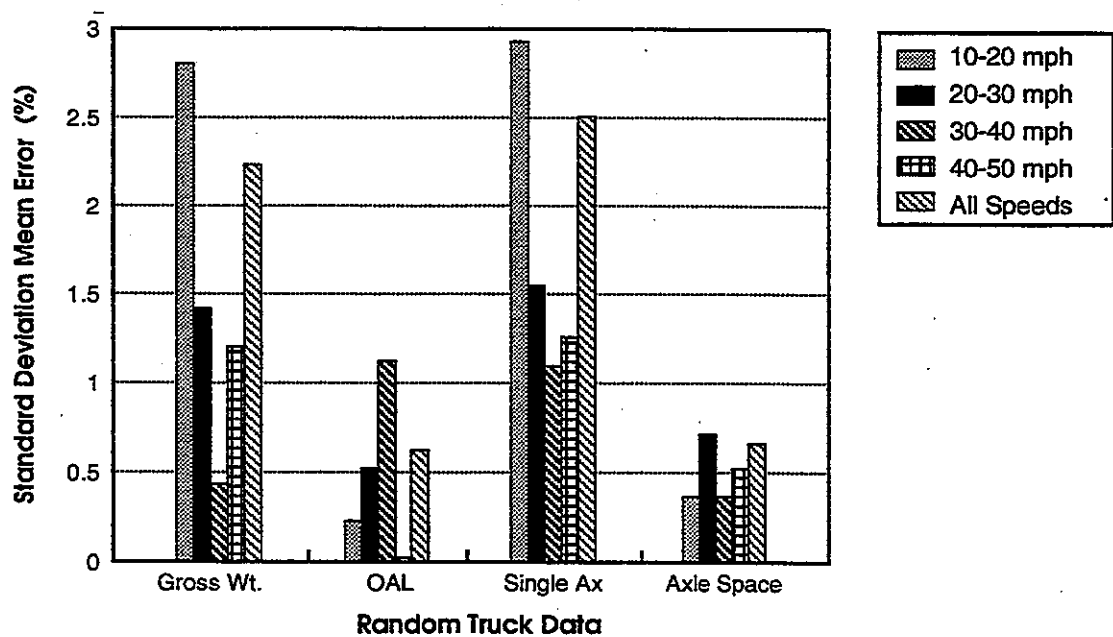


Fig. 58. **RANDOM DATA SET - STANDARD DEVIATION OF MEAN ERROR VERSUS AXLE SPACING, AXLE WEIGHT, GROSS WEIGHT, AND OVER-ALL LENGTH WITH RESPECT TO SPEED**

In general, spacing measurements were well within acceptable limits and weight errors higher than desired. Axle spacing measurements did not seem affected by acceleration or deceleration of trucks passing over the system. This acceleration-independent feature may well be due to the axle sensor array configuration designed to detect relatively instantaneous speeds. However, test data do reveal that weight measurements, among other things, appear to be very much a function of speed, as both random and controlled data sets reflect significantly greater inaccuracies for different speed groups. This observation, as noted in other WIM studies (7), is most likely due to the interaction of vehicle suspensions and pavement irregularities causing bouncing and unpredictable axle loading. However, many factors could contribute to these inaccuracies including: excessive vehicle acceleration or deceleration, roadway slope, pavement surface irregularities, vehicle suspension, wind, site geometry, and traffic volume. Slope, surface irregularities, and site geometry are the only factors that can be controlled in the system design. The Antelope site was found in a previous study (7) to have an average pavement profile index of 25.5 inches/mile, which is over 3 1/2 times greater than the department's specified limit for newly constructed PCC pavement of 7 inches/mile. This pavement profile is a measure of the smoothness of pavement surface. It has been found that pavement profile, coupled with a vehicle's suspension system, can cause considerable vehicle "bouncing", which alters dynamic wheel forces. In addition, as stated previously, an approximate 3% slope exists at the ATC/SWIM site. This figure exceeds the maximum 2% recommended by the weigh pad manufacturer. These factors, in conjunction with wide shoulder geometry, permitting trucks to leave the pavement and miss the weigh pads entirely, may significantly affect accuracy.

The prototype controller performance, in general, was more than adequate. The VME bus 3U form factor was compact and easy to work with in the cabinet. The input file interface module configuration provided a convenient means of accessing and interfacing sensor cables. Power supply problems experienced in the laboratory did not occur in the field. Some problems did occur with the VMOD digital input cards however, on occasion axle sensor signals were lost, causing

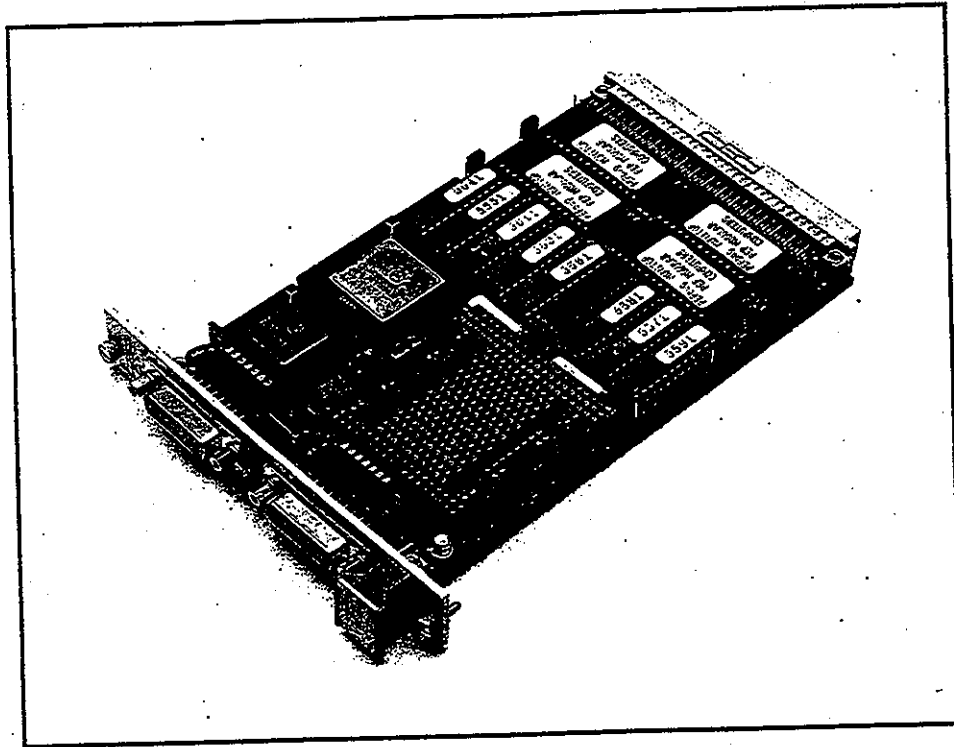
false classifications. The problem was identified as power related, and replacement of the modular piggyback card eliminated the situation.

Initially the development system was used in the field so software changes could be made easily. Laboratory software development, however, was quite successful creating little need for field alteration; some changes were made to reflect spacing discrepancies related to axle sensor locations (i.e. sensor position tables changed from 2 feet to 2.1 feet, etc.) and to calibrate weigh pads. Once accustomed to real-time programming techniques, software development was straight forward. The ability to develop in both resident and non-resident environments played a significant role in reducing development time and complexity.

APPENDIX A

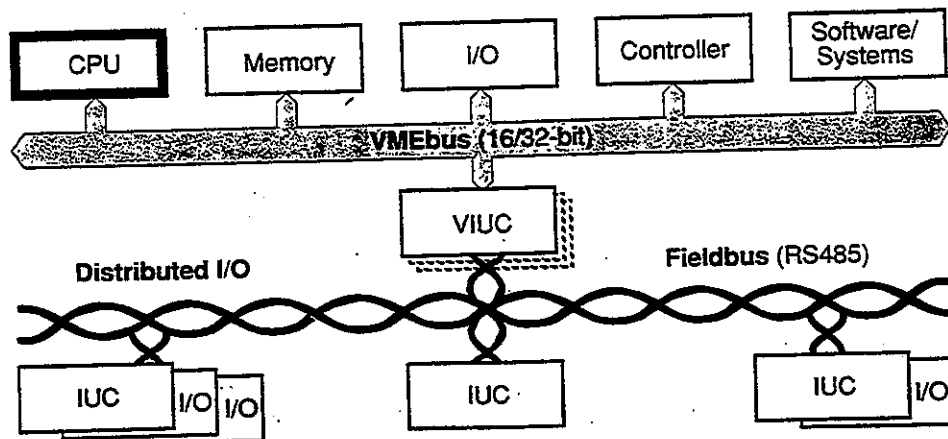
ATC Prototype Hardware Data Sheets

Although a number of hardware and software configurations are possible in advanced control applications, the ATC prototype is based on a modular design divided into five functional areas: i) standard data bus, ii) microprocessor module, iii) input/output module, iv) support hardware, and v) high-level programming language and real-time operating system software. The prototype, designed for a Slow Speed Weigh-in-Motion (SWIM) application, includes a 3U VME bus, Motorola 68020 microprocessor, RS-232 serial communication interface, TTL level digital I/O, 12-bit analog to digital converter (A/D), C mid-level programming language, and OS-9TM real-time operating system. The following section details the prototype hardware.



VM20

VMEbus Master CPU
68020/68881



Product Overview

The VM20 is a compact master CPU for the VMEbus, designed for those high performance applications where large task or mathematical calculations need to be undertaken efficiently. The VM20 is well suited for those many different single and multi-processor applications requiring: serial I/O, Dynamic RAM and large ROM ability, a real-time clock (RTC), 68020 CPU with an option for 68881 FPCP.

Features

68020 at 16.0 or 25 MHz provides the optimum of 32-bit computing power/cost. It's **Optional 68881 FPCP** (also at CPU speed) enhances this performance even further, and is of especial importance for those number intensive processes, where this highly integrated 32-bit solution on a single 4TE card leaves 16-bit solutions standing.

1, 2, 4 or 8 MBytes Dynamic Memory, provide the desired RAM and flexible tailoring for all real-time applications. The application code may be stored permanently in the "up to" 1024 KByte ROM. Both RAM and ROM are fully 32-bit organized.

Serial Communications Controller Z85C30 (SCC) provides two user configurable Serial Interfaces. Although the VM20 comes ready fitted for two RS232 ports, you may individually reconfigure either or both as desired by simply changing and fitting the appropriate Single Channel Interfaces (SC's), and thus tailoring the VM20's serial ports precisely to your needs.

Real-time Clock DP8571 (RTC), provides date and time functions as well as periodic and alarm interrupts. The RTC is supported with it's own on-board Lithium battery, which may be disconnected during long periods of storage or transport.

VMEbus Interrupt Handler and Bus Arbiter for full system controller function. SYSFAIL, ACFAIL and 16MHz SYSCLK functions are also on this compact module.

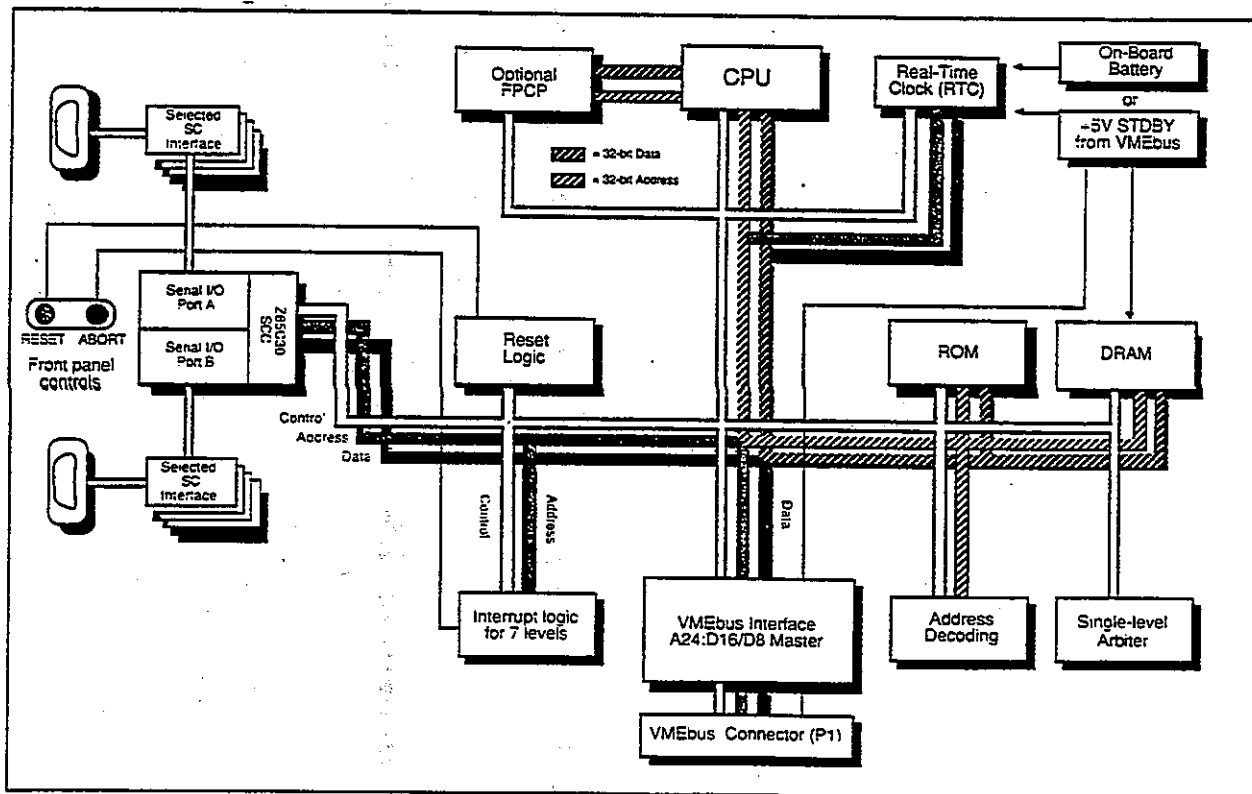
Single-height, single-slot. The VM20 is able to offer all this, including it's serial interfaces and memory piggybacks, without exceeding the standard 4 TE (single-slot) PcpCard dimensions.

Advantages

An Economical Solution: One board, many advantages, the VM20 offers full 32-bit power, fast serial interfacing with a choice of many different standards, an RTC and options with an 68881 FPCP and/or a lot of memory, for those large applications under a real-time operating system such as OS-9 or PDOS.

Flexible module tailoring. The two serial ports may be individually re-configured, to alternative I/O standards if you need, by adding the desired Single Channel Interfaces. Should your memory or interface needs change at any time, then you will only need to fit the appropriate new piggybacks.

Low power consumption of under 6.0 Watts by using CMOS technology.

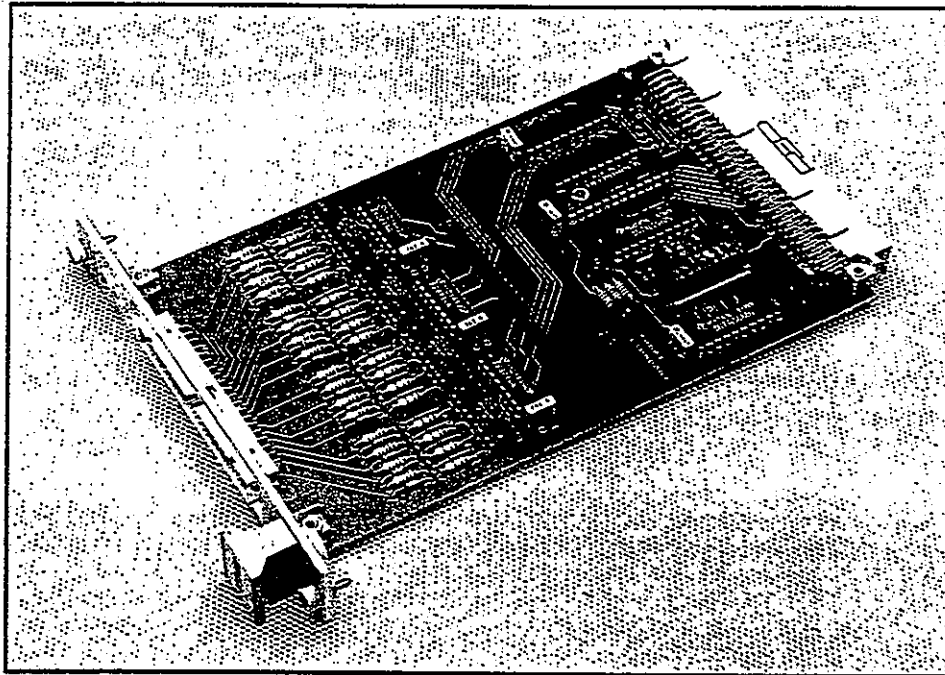


Specifications

CPU/Speed	MC68020 running at 16.0/25 MHz	Interrupt handler:	7 level static IRQ1* - IRQ7* interrupt. enable/disable and SYSFAIL* programmable via mask register.
FPCP (optional)	MC68881 running at CPU clock.	System used autovectors:	Abort switch level 7 ACFAIL* level 7 RTC(TICK) level 6 SCC level 5 (non-av) SYSFAIL* level 4
Memory:		Address modifier:	Supports Standard and Short defined AM codes.
RAM:	1, 2, 4 or 8 MBytes DRAM on a piggy-back. Option for Data Retention from +5VSTDBY VMEbus line.	Clock generation:	1 x CPU, FPCP (16 or 25 MHz) 1 x System clock (16 MHz) 1 x Refresh clock for DRAM 1 x RTC clock 1 x SCC clock
ROM:	128/256/512/1024 KBytes, 32-bit wide organisation, ≤170ns EPROM required	System controller Functions:	SYSRESET* SYSCLK* ACFAIL* (Maskable) SYSFAIL* (Maskable) Power Monitor
On-Chip Memory Cache:	256 Byte cache memory on 68020. User selectable enable/disable via-jumper setting.	VMEbus interface:	A24/A16:D16/D8. MASTER
Real-time Clock (RTC)	DP8571, 12 / 24 Hour Time-keeping. Day-of-week counter, programmable alarm and/or periodic interrupts.	Power requirements:	5V (±5%), 1000 mA typical ±12V, 30 mA typical
RTC back-up:	From on-board Lithium Battery or (on the Data Retention Versions only) from the VMEbus's +5VSTDBY line.	Temperature ranges	operating: 0° to +70°C (standard) -40° to +85°C (extended) ≠ -55° to +125°C (military) ≠ storage: -55° to +85°C
TICK:	Programmable periodic interrupt from RTC.	Operating humidity:	0 to 95% (non-condensing)
Serial I/O:	Z85C30 SCC provides 2 individually configurable serial ports, via SC-x's (Single Channel Interfaces) Ports act as DCE (Data Communications Equipment)	Board size:	Single-height Eurocard 100 x 160mm (4 x 6 1/4")
Modes of operation:	Asynchronous: 50 to 38,400 Baud Synchronous: 50 to 307,200 Baud Fully software programmable. (both modes recommended for an upper limit of 19,200 Baud under operating system control)	VMEbus Connector:	DIN 41612 style C, 96 contacts, P1 connector
Protocol:	SDLC/HDLC or asynchronous	Front panel width:	4 TE (20.3mm), 1 slot
Serial I/O SC's:	Comes configured with two SC-232's by default. Others may be ordered and fitted separately SC-232 RS232 (50 to 76,800 Bd) SC-422 RS422 (50 to 307,200 Bd) SC-GFI Fiber optic data link (including glass fiber optic interface connectors) for max. 1 km. SC-PFI Fiber optic data link (including plastic fiber optic interface connectors) for max. 50 m.	Front panel Functions:	Halt indicator (LED), Reset button, Abort button and two 15-pin sub-D sockets. Pin outs are defined by the selected SCs fitted.
Timeout function:	BERR* timeout fixed to 7.5 µs		
Bus arbitration:	Single level (BR3*) with daisy-chain logic (30 ns).		
Arbitration protocol:	Release When Done (RWD).		

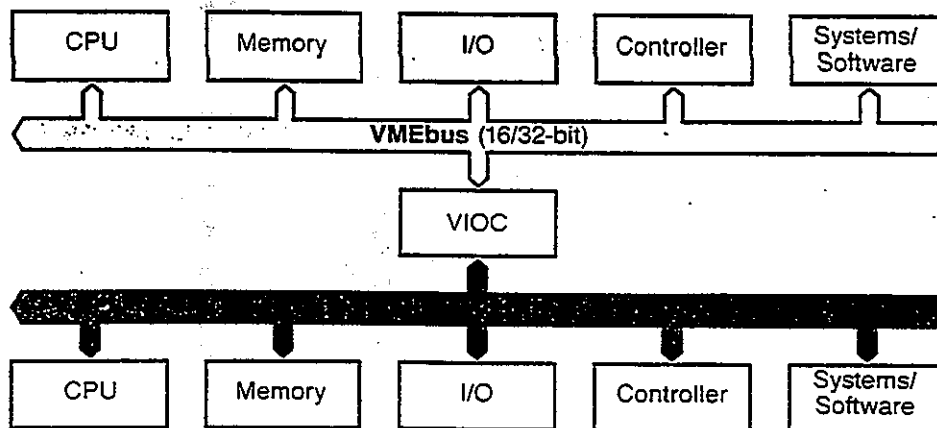
* Active low signal

= Battery must be removed when using VM20 in applications with temperatures under -25° or over +85°C.



VDIN

**16 channel Optoisolated Digital
Input Module for VMEbus**





Product Overview

The VDIN is a simple Input module for the VMEbus with sixteen individual and electrically separate optoisolated channels.

The module was designed for easy and cost-effective solutions to the many simple interface tasks which commonly arise in most industrial applications. The optoisolation offers the highest degree of protection for your valuable system and process.

Features

The features of the VDIN module include:

- Sixteen optoisolated inputs each having their own signal and return lines for your interconnection convenience.
- Full industrial input voltage range up to 24 Vdc (max.).
- Data polling for easy programming and use.
- Optional LED to display actual input line status.
- Single-height Eurocard occupying only one slot in your VMEbus system.
- VMEbus Slave interface A16:D16 with Standard/Short Supervisory or Non-privileged access.
- 40-pin flat cable connector through frontpanel for quick and easy connections.

VDIN Advantages

When using the VDIN, your advantages include:

An easy to use and cost-effective VMEbus system, the VDIN allows for the fastest development of a VMEbus system with a requirement to monitor or accept data from many different sources. No special software adaptation is necessary. The VDIN is a ready to use, ready to work PEP VMEbus product. In other words, "the right choice".

Optional version with LED line status indicators enables quick and easy monitoring or fault finding of the monitored processes. This feature (viewable through front panel) not only helps your development phase, but benefits your service people in the field, and even your customers.

Complementary VDOUT module supports applications that require output features with similar ground rules as the VDIN.

Specification

Input:	16 Channel Optoisolated
Input Voltage:	max. 24 Vdc
Input Current:	max. 10 mA at 24 Vdc
Switching Level:	< 10 V = low, ≥ 14 V = high
Propagation Delay:	typically 5 ns
Isolation voltages:	5000 Vrms, Input to System 100 Vdc between inputs
LED line status:	16 LED's show line input status
Address modifier:	Standard/Short Supervisor Data 3D/2D Non-privileged Data 39/29
VMEbus interface:	A16:D16 slave
Power requirements:	± 5 Vdc ($\pm 5\%$) 180 mA typical
Temperature ranges	
operating:	0° to +70°C (standard) -40° to +85°C (extended)
storage:	-55° to +85°C
Operating humidity:	0 to 95% (non-condensing)
Board size:	Single-height Eurocard 100x160 mm (4x6 1/4")
VMEbus connector:	DIN 41612 style C. 96 contacts, P1 connector
Front panel features:	LED display next to connector
Front panel width:	4 TE (20.3 mm) 1 slot
Front panel connectors:	40 pin flat cable connector

* The LED display is only fitted to VDIN with order No. 516-01

Ordering Information

Product	Description	Order No.
VDIN	VMEbus 16 channel input, 24 V d.c. optoisolated	516-00
VDIN	VMEbus 16 channel input, 24 V d.c. optoisolated with LED line status display	516-01

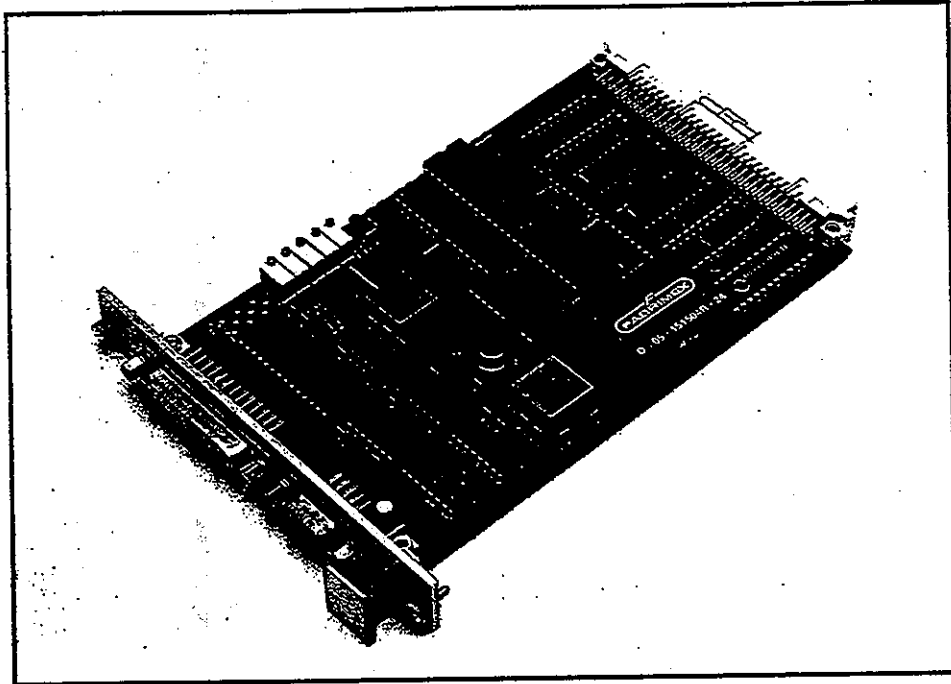
Modular Computers®

PEP Modular Computers, Inc.
600 North Bell Avenue, Pittsburgh, PA 15106
CALL 1-800-228-1737, 1-800-255-1737 (inside PA)
Telex 825098, Telefax 412-279-6860

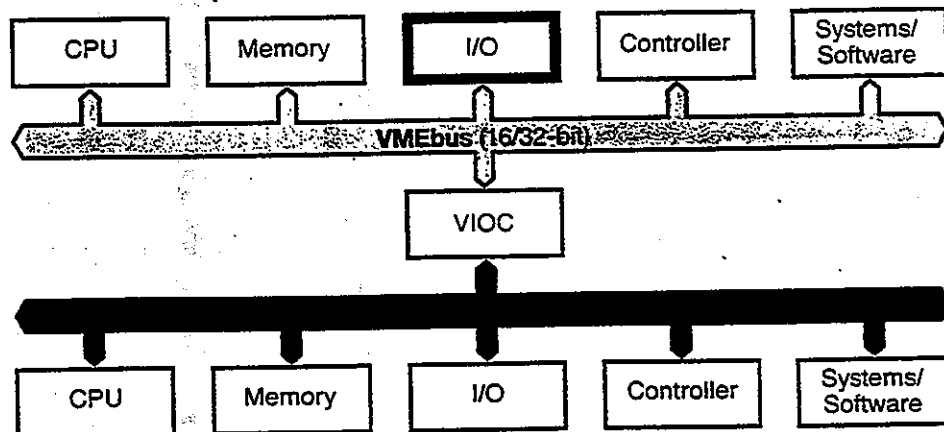
EuroPEP France
5 Rue Pierre Midrin, 92310 Sèvres, France
Telex 631335, Telefax 1-4507 1234
Phone: 1-45346060

PEP Modular Computers GmbH
Am Klosterwald 4, D-8950 Kaufbeuren, Germany
Telex 541233, Telefax (08341) 40422
Phone: (08341) 81001

PEP Modular Computers AB
Box 430, 18324 Täby, Sweden
Telefax 8-7326310
Phone: 8-7567260



VDAD Combined DAC, ADC and Digital I/O Module for the VMEbus



Product Overview

The VDAD, D to A and A to D converter for the VMEbus, is a single-height PepCard offering maximum flexibility and performance to connect a variety of analog I/O devices to your VME system.

The VDAD is a "3-in-1" PepCard, having:

- a true four channel, 12-bit, D to A converter,
- a sixteen channel, 12-bit, multiplexed A to D,
- eight Digital I/O lines for control or monitoring, and three handshake lines (for external interrupts, etc.).

Additionally the VDAD has the following facilities:

- an on-board BiTE (built in test equipment) facility to allow easy self calibration, or accuracy checks to be undertaken at any time.
- a software selectable trigger source (from either an external or the on-board 24-bit timing generator). This allows several VDADs to be synchronized, if desired.
- an optional ordering selection, with tailored board component population, for either A to D + digital, or D to A + digital, or fully populated.
- Individual Analog to Digital and Digital to Analog front-panel connectors, and careful positioning of the components keep analog signals from interfering with each other.

Features

The features of the VDAD include:

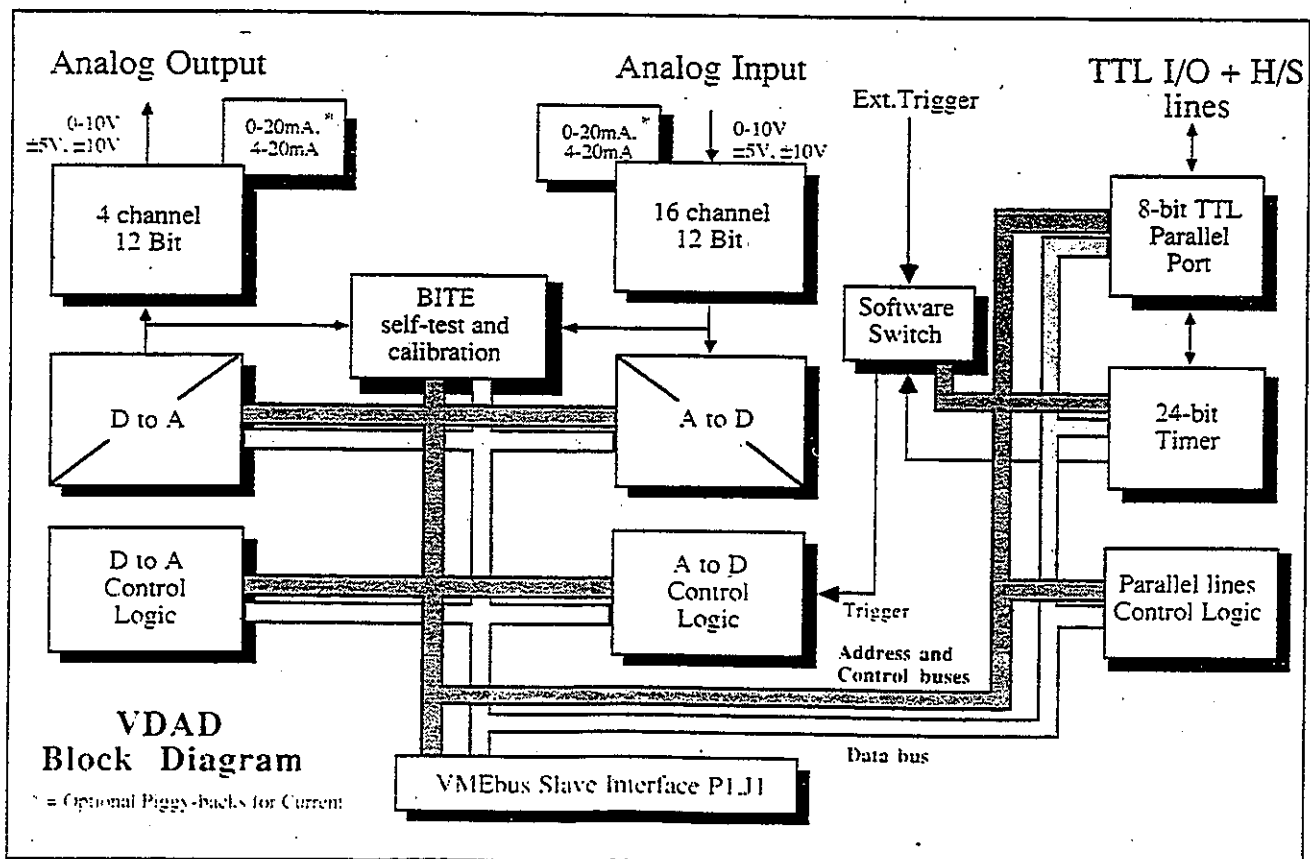
4 channel Digital to Analog converter each with independently programmable output characteristics.

- Software programmable 0-10V, or $\pm 5V$, or $\pm 10V$ outputs, or by fitting an appropriate piggy-back analog current of either 0-20mA or 4-20mA current outputs.
- 12-bit resolution, per channel
- 10 μ s settling time

16 channel multiplexed Analog to Digital converter with user definable input characteristics.

- 16 single-ended or 8 dual differential analog inputs
- Jumper selectable 0-10V, or $\pm 5V$, or $\pm 10V$ inputs, or by fitting of an appropriate piggy-back, analog current of 0-20mA or 4-20mA current inputs.
- Software programmable amplification factors (1, 10 and 100)
- 12-bit resolution, per channel. Conversion time 25 μ s standard version, or 8 μ s enhanced version.

8 Digital I/O lines with mixable input/output selection, and 3 handshake lines, these digital I/Os may be used for control and monitor (or interrupt) purposes to support the analog task to which the VDAD is being applied.



Built in Test Equipment (BITE) The VDAD has its own on-board calibration and offset correction facility built in.

The BITE may be used at any time during operation, since it is fully self-contained and its operation is entirely software controlled.

This ensures that your VDAD maintains its accuracy under the most demanding conditions, and removes the need for any additional voltage references, test routines and any additional test equipment, etc.

Benefits

When using the VDAD, your benefits include:

Flexible, cost-effective interface: VDAD's simplicity and completeness, with its built in test equipment make it a cost-effective solution when an accurate analog measurement and/or control task is required, you no longer need additional external references or complex calibration routines.

Industrial standard voltages: When applying the VDAD to any industrial control units, you have a choice of the most common industrial voltages your I/O interfaces need. Selectable via software and/or jumpers.

Industrial standard connections: the VDAD uses D-sub standard industrial connectors, the A to D connector is fully pin compatible to Analog Devices signal conditioning modules to make your connection task really easy.

Minimal system interconnections by combining all three functions, D to A, A to D, and digital I/O, in one card, your system configuration and cabling task are greatly simplified.

Low power consumption of only 3.0 Watts using CMOS technology, resulting in less heat, an improved reliability, and longer life for your system.

Single-height PepCard format provides you with the ruggedness, reliability, and compact size that is essential in harsh industrial environments.

Specifications

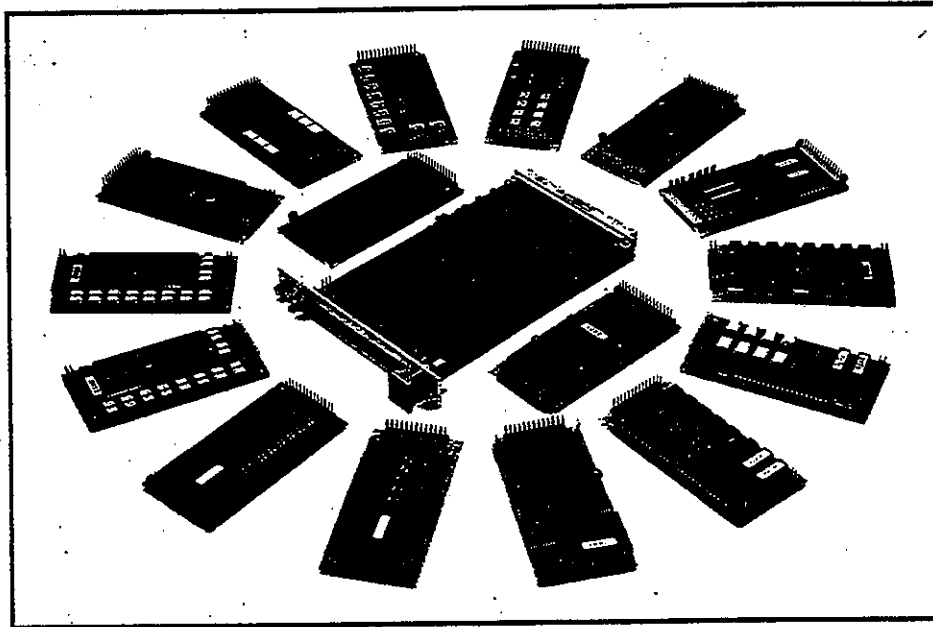
A to D converter:	HI 574 (25 μ s) Standard or HI 774 (5 μ s) Enhanced
Resolution:	12-bit
Linearity:	± 1 -bit
No. of Channels:	16 unipolar, or 8 bipolar, software programmable
Input Voltage range:	Jumper selectable for 0 to 10V, -5 to +5V and -10 to +10V
Input Impedance:	Better than 10M Ω -6pF-chn.
Gain:	1x, 10x, and 100x, software programmable. Jumper selectable for 1000x
Throughput rate:	25kHz standard, 50kHz enhanced

Specifications Continued

D to A converter:	AD6648E
Resolution:	12-bit
Linearity Error:	better than half an LSB
No. of Channels:	4
Output Voltage range:	Software programmable for 0 to 10V, -5 to +5V and -10 to +10V
Output current:	Max. 5mA/channel
Load Resistance:	Min. 2 k Ω
Load Capacitance:	Max. 500 pF
Settling Time:	10 μ s
Digital/Timer:	PI/T MC 68230, 8MHz.
Electrical levels:	High >2.0V/-100 μ A, Low <0.8V/+2.4mA
Timer Input Freq.:	Internal 8MHz, external 0 to 8MHz
No. of digital lines:	8 user-mixable-direction, plus 3 handshake (interrupt) lines
Interrupt requester:	Single level programmable: IRQ 1 - 6, non-autovector.
Interrupt sources:	Software programmable vector and level, for EOC from A to D and from digital handshake lines.
Address range:	256 Byte, may be jumper selected within the entire 16 MB address range
Address modifier:	Standard/Short Supervisory / User data access 3D/39H 2D/29H
VMEbus interface:	A24/A16:D16/D8, Slave
Power requirements:	5V dc ($\pm 5\%$), 700 mA, typical without Piggy-backs
Temperature ranges operating:	0 $^{\circ}$ to +70 $^{\circ}$ C (standard) -25 $^{\circ}$ to +85 $^{\circ}$ C (extended)
storage:	-55 $^{\circ}$ to +85 $^{\circ}$ C
Operating humidity:	0 to 95% (non-condensing)
Board size:	Single-height Eurocard 100x160mm (4 x 6 1/4")
VMEbus Connector:	DIN 41612 style C, 96 contacts, P1 connector
Front panel width:	4 TE (20.3mm), 1 slot
Front panel Connector(s):*	9-pin D to A, and 25-pin A to D sub-D mounted on front-panel

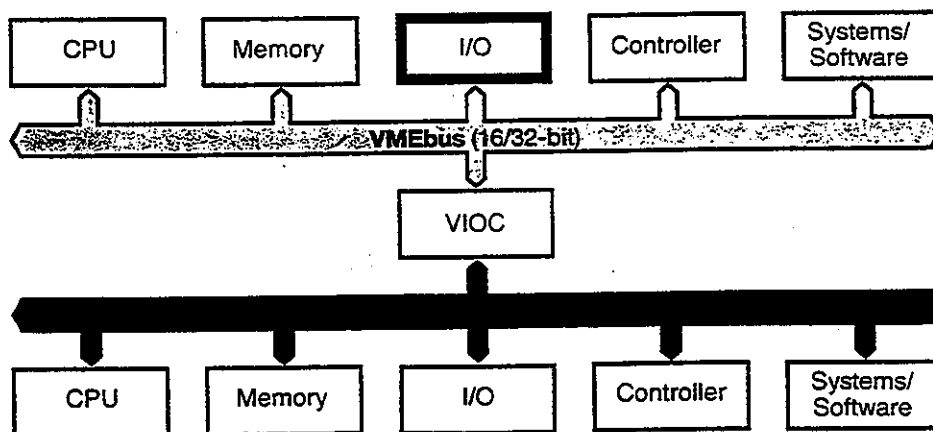
* Depends on version ordered

An optional 6U front-panel permitting the VDAD to be used in double-height systems is available on request.



VMOD-2

Flexible Industrial I/O
Interface Module
for VMEbus



Product Overview

The VMOD-2 is an improved version of the single height, VMEbus "Modular" PepCard, for versatile industrial I/O configurations, all realizable with the same adaptable base-board. It is downwards compatible with the original VMOD, accepting its piggybacks and using the same I/O connector pinouts, etc. In addition the VMOD-2 supports several new control lines to support new (and future) piggybacks. An additional circuit provides you with an "emergency stop" like control loop via two previously unused pins on the 50-way front panel connector. Since the pins used for new piggyback and front panel functions were previously left "not used" on the original VMOD, interchangeability connection problems have already been eliminated, in line with PEP's "modular" concept.

VMOD-2 may be useful in all applications requiring special motor regulators or interfaces with industrial peripherals, user definable I/O configurations, or as a simple axis controller. You may choose to fit any two industrial I/O piggybacks to your VMOD-2 module, either 2 identical or mixed, from an ever growing range of analog I/O, digital I/O, serial I/O and counter inputs, as well as motor controllers, relay devices and many other often needed "Industrial" functions.

Its piggyback configurable I/Os provide not only a cost-effective solution for a wide variety of applications, but also the ability to tailor the VMOD-2 to precisely fit your input/output needs. Only a few details of each piggyback can be listed in this data sheet. Since new piggybacks are being added to this range constantly, we advise you ask for detailed data sheets for the piggyback(s) you may require and/or for details of the latest additions to the range. A "prototyping" piggyback is also available to make your own "one-off" or "low-volume" specials, increasing the user-friendly adaptability of the VMOD-2's modular approach even further.

Features

- Accepts any two industrial I/O piggybacks from the VMOD and/or enhanced VMOD-2 PB range. (See table on next page for PB overview).
- Full 8/16-bit databus, plus all necessary and extended control lines to piggyback.
- Now 11 address lines are provided to each piggyback, for comprehensive decoding.
- Separate IRQ* lines for each piggyback
- External reset facility which can be used for local reset (or kind of "Emergency-Stop" facility through a "normally-closed" loop) on VMOD-2, and resets both the VMOD-2 and its piggybacks.
- Interface flexibility through choice of connection configurations, including Industrial Screw terminal block accessory panels.
- Full VMEbus slave Interface, A24:D16/D8 or A16:D16/D8, Slave
- One-of-Seven Jumper selectable interrupt levels
- Supports "Address Pipelining".

- Extended temperature range options, allows VMOD-2 + piggybacks to be fitted where you want them, right in the harshest of environments.
- Extended documentation and Prototyping piggyback available for designing customized Piggyback Interfaces, and your own low volume specials. Production licenses also available on request.

Specifications

VMEbus Interface:	A24:D16/D8 or A16:D16/D8, Slave
Address Range:	256 Byte, or 8KByte, block selectable. A1 - A11 available to both piggybacks for additional decoding use. Base address jumper selectable.
Address Modifiers:	Standard Supervisor/User Data or Short Supervisor/User Data access via jumpers.
Interrupt Request:	IRQ 1-7, jumper selectable. Interrupt vector generated by the piggybacks or via jumper settings.
Local Reset:	Two-wire input available on 50-way connector which may be enabled by jumper setting.
DTACK Generation:	Is generated by each of the two fitted piggybacks.
Power Requirements:	+5Vdc 140mA typical without piggybacks. Additionally ± 12 Vdc (not used by the VMOD-2) may be needed by certain piggybacks.
Temperature Ranges; Operating:	0 to +70°C (standard) -40 to +85°C (extended) -55 to +125°C (military)
Storage:	-55 to +85°C
Relative Humidity:	0 to 95% (noncondensing)
Board Size:	Single-height 100 x 160mm (approx. 4 x 6 1/4") PepCard
VMEbus Connector:	DIN 41612 style C, 96 contacts
User I/O Connector*:	50 pin flat ribbon connector with retainor/ejector latches" or 50 pin flat ribbon header".
Front Panel Width:	4 TE (20.3mm) 1 Slot

* Subject to version ordered.

Versions with front panel connector do not have the on-board header, and vice-versa. However, the pin-outs of the VMOD-2's front and on-board connectors are identical so allowing easy changes of VMOD-2 types if needed. For example, when building an internally wired target system from parts of your development system/prototype.

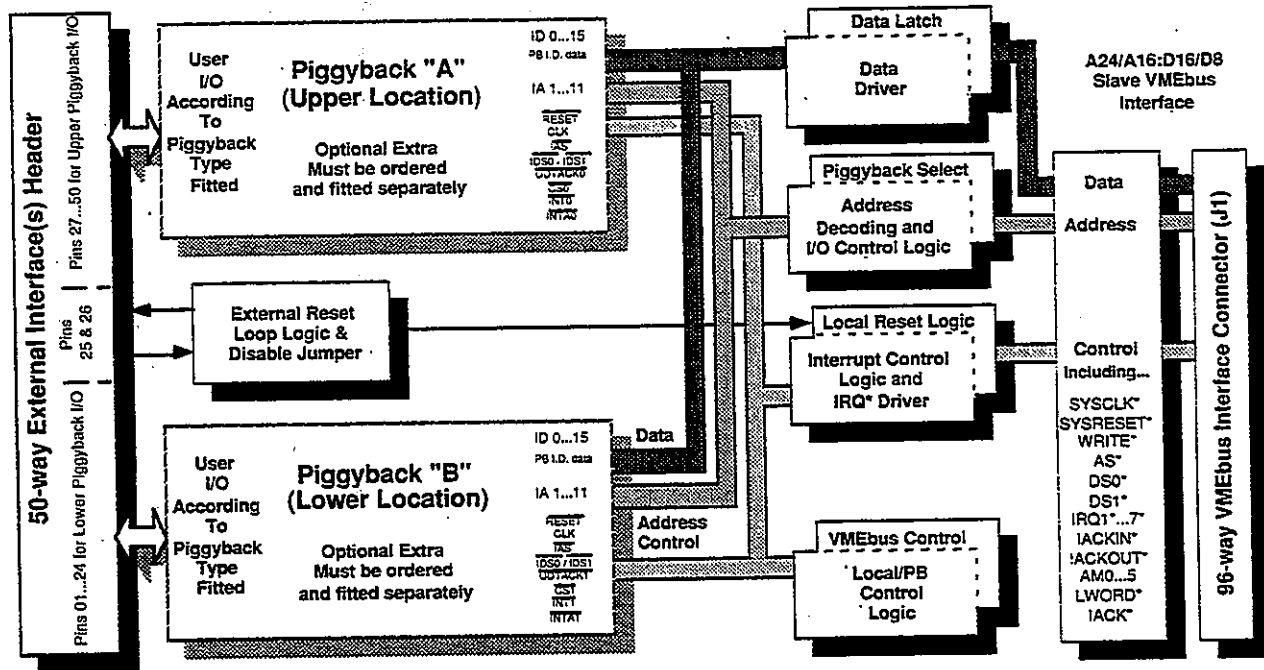
VMOD/VMOD-2 Piggyback Overview

As can be seen from the photograph on the front of this datasheet, the VMOD-2 already has an impressive range of piggybacks to choose from. This range is being continually expanded and so separate datasheets are available to cover piggyback groups and/or new releases. The table below helps provide an overview but where the data in the separate datasheet differs, the latter is to be assumed the more up-to-date. Variations (where available) are given in italics or braces, and the normal text is applicable to both. Note that 523-xx numbers are for VMOD and VMOD-2, whereas 5230-xx numbers are for VMOD-2 only.

PB-Name	Brief Description	Ch. @ V In	Ch. @ V Out	Order No.
PB-DIO	20 Ch. Digital I/O with 68230 and 24-bit timer	10 ch. at 5V / 10mA opto, Cmn Vcc	10 ch. at 5V/10mA opto, Cmn Ground	523-13/1
PB-DIO2	20 Ch. Digital I/O with 68230 and 24-bit timer	10 ch. 24V / 5mA opto, Cmn Vcc	10 ch. 24V/100mA opto, CG (CV)	523-16 (523-16/1)
PB-DIO3	20 Ch. Digital I/O with 68230 and 24-bit timer	10 ch. 24V / 5mA opto, Cmn Gnd	10 ch. 24V/100mA opto, CG (CV)	523-23 (523-23/1)
PB-DIO4	16 Ch. High Voltage Digital I/O	8 ch. 12 to 80V / 5mA opto CG in pairs	8 ch 5 to 80V/500mA opto OC CE in pairs	523-27
PB-DIN	20 Ch. Digital Input 68230 and 24-bit timer	20 ch. 24V (5V) 10mA opto CV	—	523-14 (523-14/1)
PB-DIN2	12 Ch. Hi-V Digital Input	12 individual ch.s 12 to 60V 5mA	—	523-24
PB-DOUT	12 Ch. High Voltage Digital Output	—	12 individual ch.s 5 to 80V/500mA	523-25
PB-CIO	20 Ch. "Change of State" Z8536 Inputs	20 (18) CV ch.s opto 24V/7.5mA	(2 independant ch. 24V/5mA opto)	523-19 (523-19/1)
PB-CNT	2x32-bit or 4x16-bit Counter, @ 500 kHz max. input speeds.	2/4 opto-isolated counter inputs, 24V/ 5mA (5V/10mA) [12V] [15V]	—	523-12 (523-12/1) [523-12/2] [523-12/3]
PB-SIO4	Quad Serial I/O 68681 RS232 + RTS and CTS	4 x RS232, <i>non-opto</i> (opto)	4 x RS232, <i>non-opto</i> (opto)	523-15 (523-15/1)
PB-STP	Single Axis multi-mode Stepper Motor Controller.	6 control lines @ 24V (5V) [12V] / 11mA opto-isolated	10 lines / 1 Axis 24V (5V) [12V] 8mA opto-isolated	523-22 (523-22/1) [523-22/2]
PB-REL	Eight SPDT Relays	—	8 x galv.-isolated	523-26
PB-DAC	4 ch 12-bit D to A Converter (10 μ s)	—	4ch. 0-10V \pm 10V (0-8.192V \pm 8.192V)	523-11 (523-11/1)
PB-DAC2	4 ch 12-bit D to A Converter (10 μ s)	—	4ch. 4-20 mA (0-20 mA)	523-17 (523-17/1)
PB-ADC (PB-ADC-2)	8 ch 10-bit A to D Converter (16 μ s)	8 ch. 0-10V \pm 10V (0-20 mA)	—	523-28 (523-28/1)
PB-BIT	BITBUS™ Communications Controller	80C152A. 12 (16.67) MHz. (2.4 Mbaud Sync.) 1.5 Mbaud self-clocked. 2 x 1 KByte FIFO	—	5230-11* (5230-11/1)*
PB-PRM	Prototyping	User definable functions and I/O according to your own design	—	523-18

In the above table OC = Open collector, CE = Common Emitter, CV = Common Vcc, CG = Common Ground, and opto = Opto-isolated (* = PBs suitable for use with VMOD-2 only)
BITBUS is a registered Trade Mark of the Intel Corporation.

VMOD-2 Block Diagram



Ordering Information

Product	Description	Order No.
VMOD-2*	VMEbus modular industrial I/O interface-module with 50-way front panel connector. (no on-board 50-way header fitted)	5230-0
VMOD-2*	VMEbus modular industrial I/O interface-module with 50-way on-board header. (no 50-way front panel connector fitted)	5230-1

* The VMOD-2 comes without any piggybacks fitted, these must be ordered and fitted separately as required.
VMOD-2 is also available with a 6U front panel.

PEP Modular Computers Inc.
Pittsburgh, PA 15106, U.S.A.
Telex 825098, Telefax 412-279-6860
☎ 1-800-228-1737, (toll free outside PA).
or 1-800-255-1737 (toll free within PA).

EUROPEP France
F-92310 Sèvres, France
Telex 631335, Telefax (1) 4507 1234
☎ (1) 4534 6060

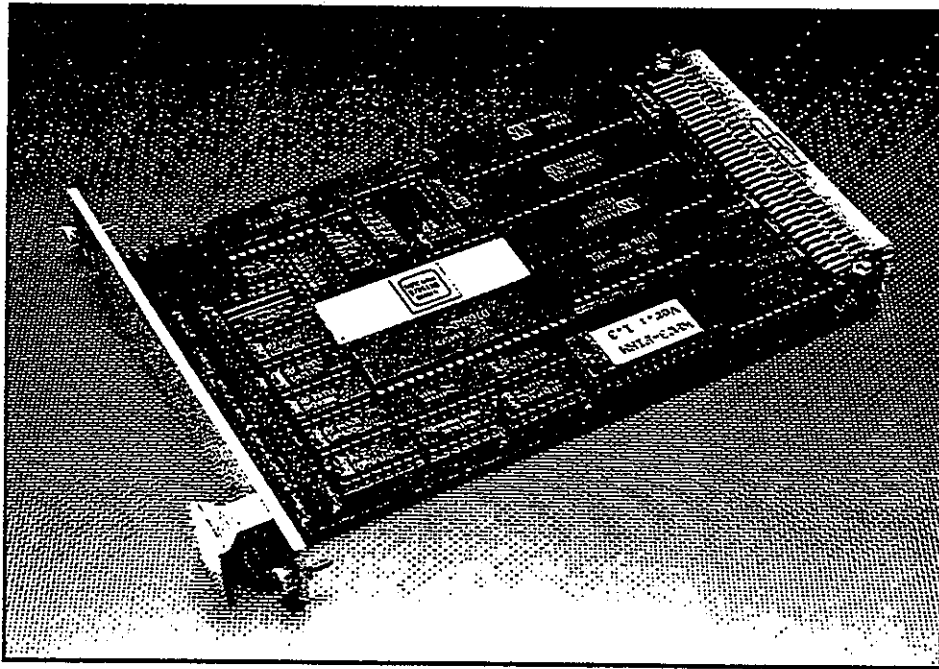
PEP Modular Computers U.K.
Portslade, BN4 1 WA, Great Britain.
Telefax (0273) 423 990
☎ (0273) 423 915

PEP Modular Computers GmbH
D-8950 Kaufbeuren, Germany
Telex 541233,
Telefax (08341) 4302-39
☎ (08341) 4302-0

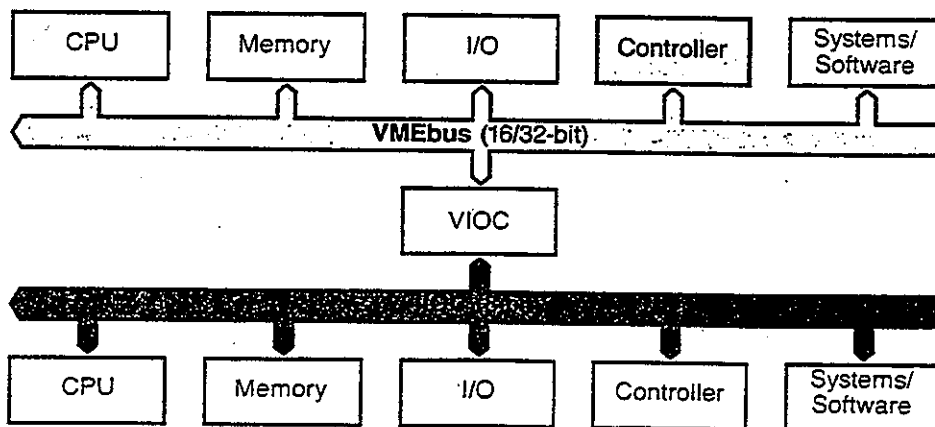
PEP Modular Computers AB
S-183 14 Täby, Sweden
Telefax (08) 732 6310
☎ (08) 756 7260

PEP Modular Computers Benelux
B-1020 Brussels, Belgium.
Telefax (02) 478 03 22
☎ (02) 478 34 16

For more information, please contact
your nearest PEP representative:



VMSC VME Intelligent Mass Storage Controller



Features

- VME Single High Eurocard
- Supports Two ST506 Compatible 5 1/4 inch Winchester Disks. Up to 5 Heads, 1024 Cylinders
- Supports Four Shugart Compatible 3 1/2, 5 1/4, 8 inch Floppy Disk Drives with Mixed Operation
- Supports One 1/4 inch Streaming Tape Drive instead of One of the Four Floppy Disk Drives
- Single and Double Density Formats
- Controls Single and Double Sided Disk Drives
- Programmable Precompensation
- Z80 CPU, 16 KByte EPROM, 16 (48) KByte RAM
- On-board Z80 DMA with 1 MByte/s Transfer Rate
- 128 Byte FIFO (Z8000) for VMEbus Transfers
- High-level Command Package
- High-speed Backup with No Host Intervention
- Head-Load Timer, Motor-On Timer
- Auto-Disk-Select for All Formats
- Automatic Bad-Track Handling
- Intelligent LRU-Buffer Control

Description

The VMSC is an Intelligent Mass Storage Controller Module for the VMEbus. The VMSC provides all required controlling, formatting and interface logic between the VMEbus and up to two hard disks, four floppy disk drives or three floppy disk drives and one streaming tape drive.

The actual strength of the unit results from the combination of powerful hardware with intelligent firmware. Accesses to the VMEbus are reduced to a minimum resulting in high efficient systems with a maximum of throughput.

High speed transmission

To avoid a loss of the speed advantage of the hard disk, the VMSC-module is directly connected with the VMEbus. A Z80 FIFO and a Z80 DMA take care for a maximum of performance. The internal 16 KB RAM (optional 48 KB) with own »Last-Recently-Used« buffer control makes the VMSC module once more quite a bit faster.

LRU-buffering shortens the response-times

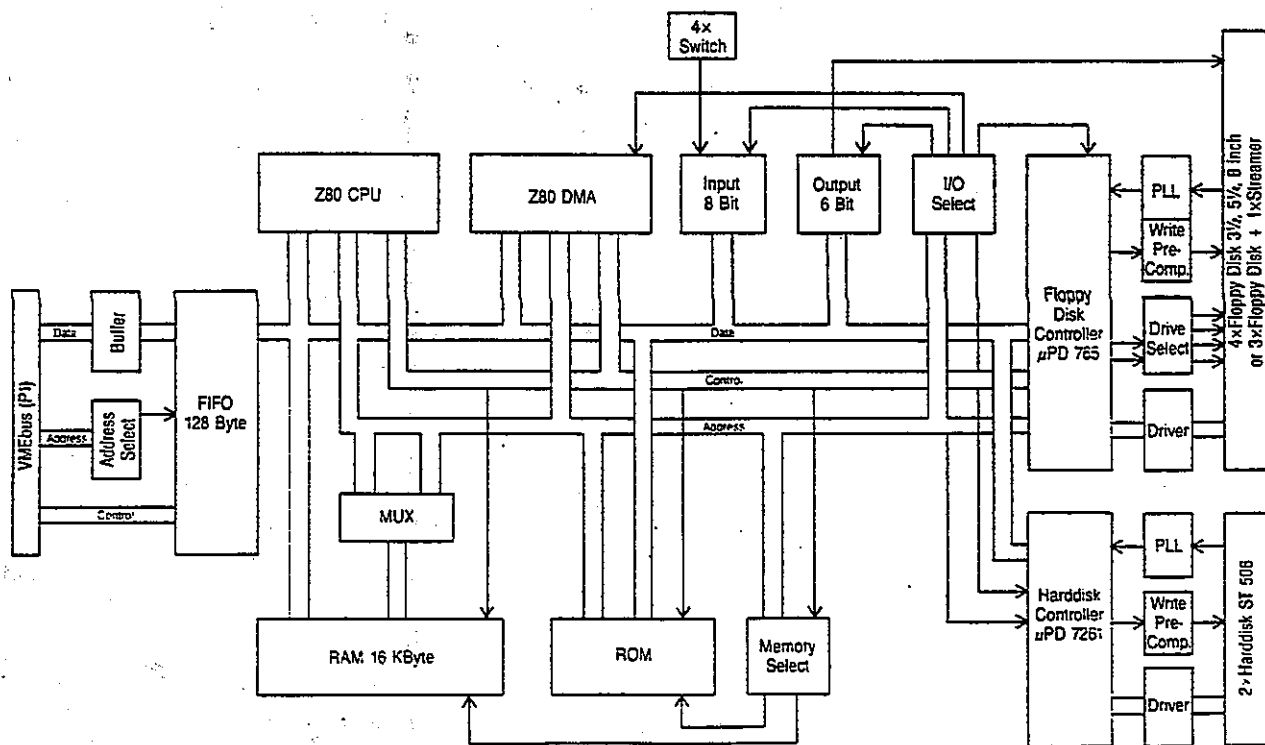
The memory of the VMSC-unit is intelligently controlled as LRU-buffer (least recently used). The generation of large buffers on track- or block-basis shortens the response time considerably as large data can be moved from or to the data media with a single access.

Blocking/Deblocking simplifies the operation

Seen from the host system, each medium whether it is a hard-disk, streamer or floppy-drive, consists of continuous 128-Byte records. It is not necessary anymore to care about the physical characteristics of the media. A for all drive types identical way of addressing on record basis standardizes the external software. Operations such as »Seek« belong to the past.

Self-diagnosis and error messages increase transparency

The VMSC firmware indicates 48 various error types. At system start the VMSC firmware performs an extensive self-test which detects and indicates faults by various



blinking periods of the front-LED. When irreparable faults occur on the medium while operating, beside the exact error type and the location a detailed error message can be read by the software in ASCII format.

Auto-Disk-Select recognizes all formats

At the access onto a medium users need not care for the type of the data medium. Mixed operation as required for some applications is self-contained. At the Disk-Login the physical format structure format is automatically detected. Thereby 36 various drive- resp. format-specific parameters are considered.

Intelligent firmware minimizes software requirements

An own Z80 processor with 16 KB firmware cares for a comfortable command set and for easiest implementation; any hardware-dependent programming work is eliminated. Beside a command port and a status register the 128-Byte deep FIFO channel effects a quick data transmission. Time consuming commands such as Back-up are executed on-board without the need for VMEbus access, reducing the bus loads dramatically.

Powerful command set

Command	Function
SETDRV	Define drive 0—5
SETREC	Define Start-Record
SETCNT	Define Record-Count
CLEAR	Clear FIFO-Buffer
READY	Test drive ready
LOGIN	Login drive
DEFDRV	Define drive
GETDRV	Get drive
READ	Read drive
WRITE	Write drive
FORMAT	Format drive
COMPARE	Compare drive
SCAN	Scan drive
VERIFY	Verify two drives
COPY	Copy 2 drives
ERROR	Get error source message
DEFBUF	Install new buffers
RDBUF	Read buffer from drive
WRBUF	Write buffer to drive
GETBUF	Get buffer content
PUTBUF	Put data to buffer
VERBUF	Verify drive against buffer
SAVE	Save all pending writes
BADWR	Write-out bad track table to drive
BADRD	Read-in bad track table from drive
BADSRC	Search for bad tracks (destroyed)
BADSRCN	Search for bad tracks (non-destroyed)
BADGET	Get bad track table content
BADLOD	Put data to bad track table
GETUSER	Get jumpers, retries, version
RWVCTR	Set r/w-retry, verify-retry, verify-control filler byte
RESET	Software reset

Specifications

Floppy/Streamer interface:

Interface: Shugart compatible
 Number of drives: 4 floppy disk drives or 3 floppy disk drives, 1 Streamer
 Floppy Tape 525CT for 1/4-inch cartridge DC600A
 Supported drives: 3.5/5.25/8 inch mixable
 Recording method: FM/MFM (single/double density)
 Data transfer rate: 125/250/500 KBit/s
 Write-precompensation: 0—625 ns, programmable
 Sector operation: Soft
 Sector length in bytes: 128, 256, 512, 1024, 2048, 4096
 Number of tracks: Up to 255
 Number of heads: 1 or 2
 Stepping method: Normal Seek
 Stepping cycle: 1—16 ms
 Pin row connectors: 50 pins for 8 inch drives
 34 pins for 3.5/5.25 inch drives

Hard disk interface:

Interface: ST 506 compatible
 Number of drives: 2
 Supported drives: All common drives
 Recording method: MFM (double density)
 Data transfer rate: 5 MBit/s
 Write-precompensation: 8 ns, determined by a PAL
 Sector operation: Soft
 Sector length in bytes: 128, 256, 512, 1024, 2048, 4096
 Number of cylinders: 1024
 Number of heads: 1—8
 Stepping method: Buffered Seek
 Pin row connectors: Data: 2×20 pins
 Control: 1×34 pins

Hardware:

Processor: Z80A, 4 MHz
 Memory: 16 KByte EPROM, 16 KByte DRAM (48 KByte optional)
 Floppy controller: μ PD 765, PLL via 9229
 Hard disk controller: μ PD 7261, PLL via 8460
 VMEbus configuration: A24:D16 Slave
 Interrupter options: I(6)
 Power requirements: +5 Vdc, $\pm 5\%$, 800 mA
 Operating temperature: 0 to 70 degree C
 Operating humidity: 5% to 95%, non condensing
 Physical configuration: SINGLE
 Physical dimensions: Single high Eurocard, 100×160 mm
 VMEbus connector: DIN 41612 style C, 96 contacts
 Frontpanel width: 4 TE (20.3 mm)

APPENDIX B

Object-Oriented Application Development Software

The Advanced Transportation Controller Software (ATCS) package is an object oriented application generator designed to allow non-programmers to develop transportation software for the ATC prototype and similar VME bus platforms. This package is coded in C and OS-9TM. It is based on a library of pre-programmed functions that handle typical traffic engineering tasks. A number of Traffic Control BLocks (TCBLK) have been developed to count vehicles, measure occupancies, archive data, change metering rates and message sign warnings, cycle traffic signals, and execute other traffic engineering functions. The Object-Oriented Application Development Software is described in the following section.

Software for Advanced Traffic Controllers¹

by Darcy Bullock² and Chris Hendrickson³

Abstract

A systematic approach to traffic engineering software development could provide significant advantages with regard to software capability, flexibility and maintenance. Improved traffic controllers will likely be essential for many of the proposed intelligent vehicle highway systems (IVHS) applications. This paper introduces a computable language, called TCBLKS (Traffic Control Blocks), that could provide the foundation for constructing real time traffic engineering software. This computable language is designed to be configured by a graphical user interface that does not require extensive software engineering training to use, yet provides much more flexibility and capability than possible by simply changing program parameters. The model is based upon the function block metaphor commonly used for constructing robust and efficient real time industrial control systems. Adapting this model to the transportation sector permits traffic control applications to be programmed by: i) selecting pre-programmed function blocks from a standard library, ii) configuring block parameters, and iii) connecting blocks to other blocks in the strategy. The software model described in this paper has been implemented in C on an advanced traffic controller platform and demonstrated in real time under simulated conditions for applications such as signalized intersection control, ramp metering, and communications with existing traffic control devices.

1. Introduction

Twenty years ago, traffic controllers underwent a technical revolution in the switch from electromechanical systems to solid state microprocessor systems. With the computing technology available two decades ago, the most cost effective approach for software development was to construct specialized, embedded systems tailored to the traffic control industry. Traffic control logic was programmed using assembly language programs that could read and write bits associated with external sensors and actuators. Initially, these microprocessor based controllers did little more than their mechanical predecessors. Over time, transportation engineers realized that more and more features could be implemented on solid state controllers and upgraded their software accordingly.

Today, we face another turning point in traffic control technology. The tremendous advances in microprocessor technology over the past decade has seen the average cost of computing drop by roughly an order of magnitude every 5 years [Rappaport 91]. In contrast, the average cost of a Caltrans 170 controller has decreased only 15% since 1987, and 27% since 1982 [Caltrans 92]. Off-the-shelf, field-

¹Prepared for the 1993 Annual Meeting of the Transportation Research Board.

²Assistant Professor, Department of Civil Engineering, Louisiana State University, Baton Rouge, LA 70803; 504/388-6826; Internet: darcy@sun-ra.rslu.edu

³Associate Dean, Carnegie Institute of Technology and Professor, Department of Civil Engineering, Carnegie Mellon University, Pittsburgh, PA 15213; 412/268-2948; Internet: cth+@cmu.edu

hardened and affordable equipment is available that rivals the computing power of mainframes from twenty years ago. If computing costs continue to decline in this manner, it will no longer be cost effective for proprietary transportation computers to compete with mass produced industrial hardware. Migration to these more powerful computers will allow traffic engineers to make a fundamental change in software development practices. Memory capacity and processor limitations will not impose significant constraints on applications. Instead, traffic engineers can focus on developing an efficient architecture for building systems that are more effective, easier to install, and easier to maintain.

This change should have numerous benefits. Since the inception of the microprocessor based traffic controller, the software engineering effort devoted to constructing traffic control software has been less than ideal [Chase 89, Bullock 92a]. In general, the state of the practice with current microprocessor software is to write software in assembly language, without an operating system, permanently install the software on a chip (i.e. burn it into a ROM) and empirically test programs to see if they work. The poor software quality resulting from these ad hoc development techniques have caused traffic engineers to be very cautious about "improving" traffic controller software. Provided a suitable programming model can be developed, we can now engineer software for greater capability, flexibility, and usefulness. However, no substantial model has been proposed. Such models are crucial for the evolution of an engineering discipline from a solely craft based practice [Shaw 90]. Imagine what it would be like to size a structural mast or electrical circuit without the model for an elastic beam or Ohm's law!

Selecting an appropriate software model is particularly important since the development of transportation control systems is multi-disciplinary, requiring the interaction of transportation engineers, electrical engineers, software engineers, and government officials. In the past, the coupling between traffic engineering concepts and field implementation has been weak. This paper presents a software model that is designed to address professional communication gaps and the need for more capable and maintainable software. It is based on the "function block metaphor" widely used in industrial control systems. This model provides the capability for non programmers to develop intuitive control software by drawing graphic diagrams on a computer screen and filling in menus. This model is based on a formal, real time scheduling algorithm that allows the correctness and feasibility of strategies to be formally verified. It has been so successful in the industrial sector that many companies have imposed rules restricting development of custom software and require application developers to use "canned software" applications consistent with the function block model.

The following section introduces as background the rationale and characteristics of an advanced traffic controller hardware platform. The three subsequent sections describe our model of traffic control software with an example, an overview, and technical details of implementation. The following section provides an example application implementing responsive freeway ramp metering. A final section discusses some considerations regarding the evolution of advanced traffic control software.

much longer it will be cost effective for the transportation community to manufacture specialized computers.

These shortcomings and requirements for improved software development tools, faster processors, expandable I/O, and more memory have led the California Department of Transportation to investigate the use of modular industrial computers for applications ill suited to the 170's [Quinlan 89]. This proposed platform, called the Advanced Traffic Controller (ATC), is based upon a 3U VME bus, a 680x0 processor, and the OS-9TM operating system. This computer is used extensively in the military and commercial sectors and provides an economical, off-the-shelf hardware platform for the ATC. Although a rich set of development tools, including operating systems, compilers, debuggers, are available for this platform, the low level nature of the tools renders them inappropriate for everyday use by traffic engineers. This is analogous to a desk top computer that only has a language compiler. For a desk top computer to be truly useful to an engineer, application software such as a spreadsheet or CAD package must be available. Due to this ATC software void, a general purpose application program is required to enable traffic engineers to develop real time traffic control strategies.

3. A Computable Language for Traffic Engineering

This section introduces, by an example, a computable language that could provide the foundation for a traffic application software paradigm. This software model could be used by traffic engineers to rapidly configure robust real time traffic control algorithms. The motivation for developing a computable language is to provide a high level configuration tool that does not require extensive software engineering training to use, yet provides more flexibility then just changing program parameters. The model underlying the "language" is based on function block programming in which the function blocks specialized to traffic engineering are graphically assembled and downloaded to a field controller. Several alternative, commercial task level programming techniques could also be used, including ladder logic, state diagrams, batch languages, and sequential charts. However, the function block programming technique was selected as the most suitable for common real time control and data acquisition problems in transportation [Bullock 92b].

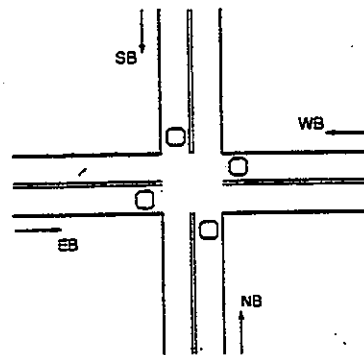


Figure 1: Example intersection.

2. Advanced Traffic Controller Hardware

Most of the current generation of traffic controllers used in the United States are based upon two different families of controllers. This section summarizes these competing efforts and describes some hardware modernization efforts underway.

One family of controllers, referred to as NEMA units, are built with connectors conforming to standard mechanical and electrical connectors. The philosophy of this standard is that manufacturers will compete based upon the hardware and software they provide inside the controller. In theory, an agency can migrate to another manufacturer's controller by un-plugging the old one and plugging in the new one using standard connectors. Due to additional proprietary sockets added to the NEMA TS1 units and non standard communication protocols, this interchangeability is not realized in practice. The 1988 NEMA TS1 standard has recently been updated (NEMA TS2 Type 1 and NEMA TS2 Type2) to address deficiencies of the NEMA TS1 standard and incorporate an alpha-numeric display for interaction with the controller. Since the software on all NEMA controllers remains proprietary and cannot be ported by the customer, engineers and technicians must still learn new software in order to reconstruct timing and phasing plans on new NEMA controllers.

A second family of controllers, referred to as the Caltrans Type 170 controllers, are built to provide both standard connectors and portable software. The philosophy of this standard is to develop a very precise specification for a traffic control microcomputer. Manufactures are selected periodically based upon competitive bidding. This standard has been tremendously successfully for the past twenty years. Minor modifications have been introduced over time, including a second serial port, additional memory, and different ROM sizes, but the essential features are unchanged. The distinguishing feature of 170 controller remains the program module, an insertable card with a ROM that stores the traffic control program. This module can be removed from one manufactures' 170 controller, inserted in another manufactures' controller, and the software will run without modification. Instead of relying on embedded user interfaces as in the NEMA controllers, the 170's are typically configured by connecting a small computer such as a PC to a serial port and down loading the strategy. Alternatively, binary configurations can be keyed in on a hexadecimal keypad. A modernization of the Type 170 has been undertaken by New York state and is called the Type 179. This controller provides more powerful computing and employs a real time operating system. However, it is still based upon a proprietary hardware standard and the software development process is subject to the same limitations of the Type 170.

In view of the tremendous microprocessor and software engineering developments in the past two decades, these standards are beginning to age [Bullock 92a]. First, the software is written entirely in assembly language. The complex nature of assembly language development precludes all but the largest cities and states from maintaining a software staff for making software configuration changes other than changing parameters in a given configuration. Second, no operating system is employed (except for the 179). Routine chores such as task scheduling, memory management, and semaphores, must be re-coded. The "home-grown" executives that have evolved preclude sharing of new control strategies. Third, the hardware constraints (slow processors, limited memory) can only be addressed by a revised standard that would require rewriting large quantities of assembly language applications. Finally, it is unclear how

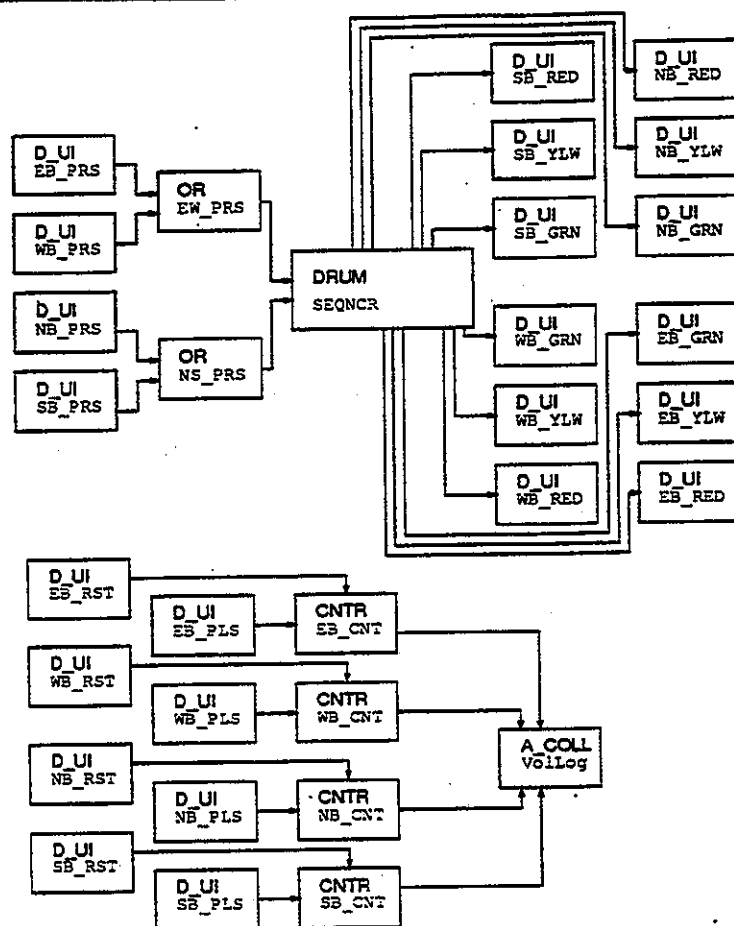


Figure 2: Semi-actuated intersection control strategy using five function block types.

The cycling of the signal is performed by the DRUM function block named SEQNCR. This block is designed to provide cycling operations similar to a mechanical drum sequencer. The conceptual operation of this block is shown in Figure 3. The generic design of the block permits the DRUM block to be used in many different cycling applications. The block has up to 11 inputs to control the operation of the virtual drum and 26 outputs that can be connected to other blocks or actuators. The block is configured by specifying maximum and minimum dwell times in each step and corresponding output states for up to 16 digital outputs. If a step hold input (HLDxx) is off, the block remains in a step only until its minimum hold time expires. If a step hold input (HLDxx) is on, the block remains in a step until its maximum hold time expires. The NEXT and PREV inputs can be used as an override to force the DRUM block forward or backward, regardless of the duration spent in a particular step. The ENABLE input is used to turn the block on and off. When ENABLE is on, the block runs in automatic mode and cycles through the steps. When ENABLE is off, the block is essentially in MANUAL mode, the outputs remain in their last state and the block does not change states (although the state outputs can be forced on

In the function block programming paradigm, a user develops applications by selecting and connecting pre-defined software modules called "blocks." The blocks represent parameterized programs prepared in a uniform manner, which permits them to be interconnected with other blocks. Connections between blocks serve as communication links for particular variables such as detector states, approach volumes, or phase timing. Selection of blocks may require definition of parameters such as evaluation frequency, minimum and maximum green extensions, and filter times. Function block programming lends itself readily to graphical displays in which blocks are represented pictorially as a box with a title, indicating the program associated with a particular block, and a name, providing a symbolic means of referring to elements of a specific block. Figure 2 shows an example. Connections, or data flows are shown as link connections between the boxes. A typical function block program resembles an activity-on-node (PERT) project management scheduling network.

The reader should be aware that function block programming is different from "modular design" taught in introductory programming classes since the end user never encounters any procedural code. All interaction with hardware devices, protocol conversions, buffers, timing demands, and error recovery are embedded in a parameterized function block *graphical icon* that can be configured by a traffic engineer using a function block editor. The blocks available within the function block editor are prepared by software engineers in a standardized manner which permit seamless interconnection and implementation.

Function block programming is analogous to a language in that control strategies (messages) are constructed by assembling blocks from an existing block library (words). Rules specifying how blocks can and can not be connected restrict the allowable juxtaposition (grammar). New blocks (words) result from the combination of old words and new inventions required to address particular needs. Based upon a relatively small set of blocks and a few connection rules, a wide range of strategies can be constructed.

To illustrate this technique, suppose we wish to configure a control and data acquisition system for the simplified semi-actuated intersection depicted in Figure 1. There are only two phases in this example, and we wish to construct a strategy that will cycle through two phases, each with a minimum green interval. If a call is received for a phase, the phase will be extended until either the call ends or the phase reaches the maximum extension time. Using the configurable function block model, a basic semi-actuated controller and data acquisition strategy could be constructed by assembling the blocks shown in Figure 2. The top portion of the strategy depicted in Figure 2 corresponds to the semi-actuated signal controller and the bottom portion of the figure corresponds to the data collection portion of the strategy.

In the semi-actuated controller portion of the strategy, the four blocks on the left belong to a class of function blocks called D_UI, short for Digital User Interface blocks. These blocks are used to read the digital inputs connected to the presence outputs on the loop detectors on the Eastbound, Westbound, Northbound, and Southbound approaches respectively. The East/West phase is extended if there is a vehicle presence on either the Eastbound or Westbound loops. This logic is performed by the OR logic block named EW_PRS. Similarly, the North/South phase is extended, if there is a vehicle presence on either the Northbound or Southbound loops. The outputs of the blocks EW_PRS and NS_PRS indicate if a particular phase should be extended.

on one of several media, including a hard drive, floppy drive, RAM drive, or non volatile disk drive emulator. The data collection block can be configured to log changes for every vehicle, to measure individual headways, or to record volumes only when they change by more than a certain number of vehicles. Also, the counts can be reset by a connection. These reset connections can be driven by any digital event, including a manual toggle of the D_UI blocks, a connection to a clock that periodically resets the counters, or a electrical contact connected to the appropriate software block.

The motivation for developing these blocks is to establish a vocabulary of control blocks that can be used by the traffic engineer to implement high level control concepts. The set of blocks is called TCBLKS for Traffic Control Blocks. It is possible to develop one exceeding complex block that would interact with I/O, log data, and sequence lights. For example, one block could emulate every single function of the California C-8 signalized intersection software. However, the function block architecture is better utilized by providing a family of function blocks dedicated to, say, three legged intersections, four legged intersections, five legged intersections, ramp metering and so on. Each of these blocks would then be configured in a manner similar to that depicted in Figure 3. Depending upon the application, the engineer would select the appropriate block, specify the relevant parameters and connect the block to the requisite I/O blocks.

4. Traffic Engineering Function Block Programming Model (TCBLKS)

The previous section introduced the function block programming model by an example. This section addresses three areas: i) Traffic engineering task vocabulary, ii) Configuration of function block strategies, and iii) Function block language structure. The following section discusses more detailed implementation issues.

Traffic Engineering Task Vocabulary. The set of building blocks available in the block library constitutes the "vocabulary" for users to assemble applications. Table 1 summarizes the forty blocks we have developed. Several of these blocks were described in the previous section, such as the Drum for signal sequencing. Also included in the block vocabulary (Table 1) are signal filters, logic functions, interfaces to external sensors and actuators, archival functions and various algorithmic blocks. The intent of establishing a definition of these control blocks is to provide a vocabulary that can be assembled by a traffic engineer (in a sketch or diagram) to define the required software. This concept is used extensively in the chemical and process engineering fields so that there is an almost one-to-one correspondence between the process and instrumentation diagram (P&ID) developed by the chemical engineer and a function block strategy constructed by the control system contractor. We seek to develop the same continuity for traffic engineering.

Since this model is only in the prototype stage, the blocks described in Table 1 currently fall short of providing a comprehensive set of building blocks. To support the growth of this model, new blocks can be created and included in the block library as long as the new blocks conform to standard block definition and operation practices. Thus, applications such as a dynamic signal control algorithm like OPAC [Gartner 83] could be included in a single function block. In general, this model supports blocks of varying execution complexity ranging from simple logic gates to complex blocks supervising several

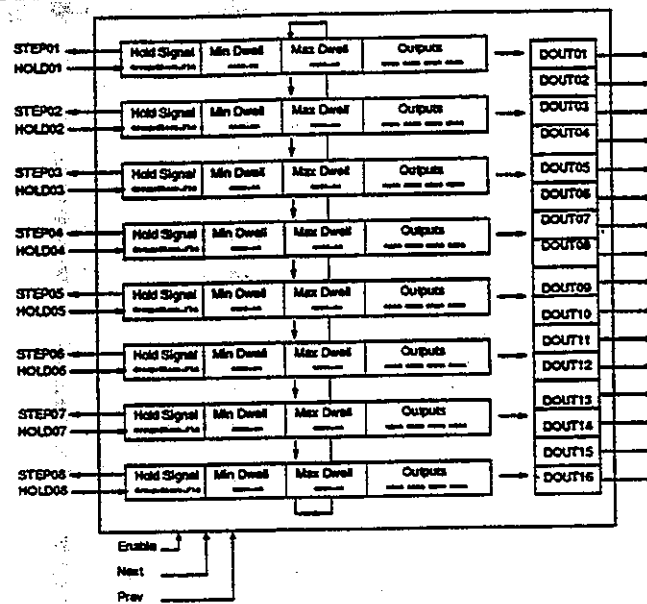


Figure 3: Internal details of a generic traffic signal drum sequencer

or off when the block is in manual). The block has 16 outputs, each of whose state is controlled by the output field set in each step. When a step is entered, all block outputs are immediately set to the state specified in the corresponding outputs field. In addition, each step has an individual output (STBxx) indicating the current step (on if current, off otherwise). For this particular block, the maximum number of steps available is eight, but a smaller number can be used. This example uses a small, general purpose DRUM sequencer with 8 steps and 16 outputs to illustrate the basic sequencing operations. In practice, larger blocks providing more steps and outputs would be used. Also in practice, a user can ignore the details of the DRUM block shown in Figure 3.

The outputs of the DRUM block are connected to the user interface blocks shown on the right side of the Figure 2. These block outputs control the electrical outputs used to turn the appropriate lights on and off. The use of these "user interface" blocks for both the inputs and outputs is an important feature of the function block model since they connect the software signals to the physical I/O and can also be put in a manual override mode by setting the ENABLE field to off. This override mode permits example inputs to be manually entered (to simulate certain actuation patterns if physical inputs are not available), or the strategy to be run without actually driving the physical lamps. Once a strategy has been fully tested, the blocks can be enabled.

This example also illustrates how easily a control strategy can be augmented with a set of blocks to collect traffic volume data. These are shown in the bottom of Figure 2. Four of these blocks read the pulse output of the loop detectors on each approach. These blocks are connected to a counter (CNTR) block on each approach that increment an internal counter on every rising edge. The output of these counter blocks are connected to a data collection block that logs the data to a file. This file can be stored

2. **CONFIGURE:** Parameters defining a program block's operation, such as the number of phases or a loop detectors I/O port, are configured for each block. This procedure is performed by selecting a block with the mouse and choosing the "Configure Parameters" option. Of course, each block is instantiated with a full set of default values that may be acceptable, in which case this operation could be omitted for many blocks.

3. **CONNECT:** The blocks are connected by clicking on a block, selecting a particular output socket, clicking on another block, and selecting a particular input socket. Basic error checking is performed to prevent sockets with different data types from being connected. For example, it would be invalid to connect the state of a loop detector to the socket determining the cycle length for a traffic light drum sequencer.

These steps are only intended to give the reader a flavor for how the function block model could be configured. In practice, these steps will likely be intertwined as a strategy is developed and edited in an incremental manner. Past strategies would typically serve as templates for new applications. Also, a number of diagnostic, reporting, drawing, scaling, and annotation tools are necessary to round out the features of the configuration tool.

Language Structure. Table 1 provides a summary of some pre-programmed blocks that can be used to develop block strategies. This section details the basic architecture of those blocks and how they can be assembled. Abstractly, a function block is a vector consisting of the following elements (Figure 4):

- **Input Sockets** which are used either to retrieve data from other blocks or are assigned constant values. Input sockets are actually references to memory locations where the block reads values from. The values stored in those locations can either be changed by another block's output socket, or by an operator manually inserting a value. These sockets represent the destination half of a data flow connection.
- **Local Storage** for storing block parameters and interim calculations.
- **Output Sockets** which are used to store block output values and can be connected to other blocks. Output sockets are actually references to memory locations where the block will write output data to. The values written to those locations can be read by another block's input socket, or by an operator examining sockets. These sockets represent the source half of a data flow connection.
- **Block Algorithm** that periodically reads the values associated with the input sockets, performs calculations, manipulates local storage, and then updates the output sockets.

Although blocks may have several input or output sockets, it is not required that they all be connected. In fact, input sockets can be assigned either constant values (Figure 5, Socket 3) or connected to another block's output socket (Figure 5, Socket 2) during configuration. Similarly, output sockets can be left dangling (Figure 5, Socket 1) or connected to input sockets (Figure 5, Socket 2) on other blocks. The only restriction on connecting blocks is that one input cannot be connected to more than one output socket (Figure 6).

ramp meters. For example, the simple blocks such as mathematical computations and digital logic are necessary for incorporating minor operational changes typically required by peculiar geometric or policy constraints. In contrast, the complex blocks such as ramp meter and intersection control can provide rapid and reliable task level programming.

AND, OR, XOR, NOT: These block perform the essential boolean logic operations on their input(s).

DlyOn, DlyOff, OneShot: These digital blocks perform digital logic timing operations. The DlyOn block delays a transition from low to high for a specified time. Alternatively, the DlyOff block delays a high to low transition for a specified time period. The OneShot provides a pulse generating mechanism for transitions from low to high.

D-Shift: provides a 16 bit shift register for transient storage of digital states.

D-UI: provides an operator with simple on/off and pulse operations for user interfaces.

Match: provides basic decoding functionality.

Timer: measures the duration of digital events.

Counter: can be used for counting lo-high transitions.

FF-RS, FF-D, FF-JK: These blocks provide discrete implementations of clocked RS, D, and JK flip flops. A T flip flop can be constructed from the JK flip flop.

Drum: provides state sequencing subject to minimum and maximum durations with the capability of back stepping.

Rate: calculates the filtered rate of an incoming digital pulse train.

Add, Mult, Div: These block provide basic mathematical operations.

Mavg, A-Shift: Both blocks implement a circular queue. The Mavg block uses the queue to compute the moving average of a time series. The A-Shift block provides a mechanism for introducing a time delay (lag).

A-Latch: latches an analog value when a digital pulse is received.

A-SWCH: selects between two analog signal based on the state of a digital input.

A-UI: provides an operator with a mechanism to enter an analog value for a user interface.

Filter: provides a simple discrete approximation for a first order analog filter.

Test: compares an input against a set of absolute Hi and Lo bounds or relative to another signal. The results of these comparisons are digital points other blocks can connect to. It is useful for implementing conditional logic.

Sel-H, Sel-L, Sel-M: High, low and middle selector blocks. The first two blocks have two inputs, the middle selector requires 3 inputs.

RMSB: provides supervisory rate selection of a ramp metering rate based upon one upstream volume sensor and up to six downstream occupancy values.

LOOKUP: provides a interpolated lookup table for defining non-linear transformations.

D-Coll, A-Coll: monitors up to eight inputs (Analog or Digital) and records their state to a file. A background spooler is set up so this file can reside on any OS-9 file device. These devices include hard disks, floppy disks, RAM disks and non volatile disks.

RMDI, RMDO: used to read digital inputs (DI) or write digital outputs (DO) on a 170 running ramp metering software.

RMRI, RMRO: used to read register inputs (RI) and write register outputs (RO) on a 170 running ramp metering software.

VMS: contains up to 8 prioritized messages that can be displayed by triggering a digital input.

Table 1: Function Block Summary

Configuration of function block strategies. An advantage afforded by the function block programming model and advocated in this paper is the ability to easily "program" or configure robust traffic control software without an extensive software engineering background. A typical configuration tool can operate like a simple vector drawing package commonly found on desk top or notebook computers. Instead of manipulating shapes and lines, it manipulates function blocks. A block program is developed by assembling a "strategy" composed of pre-defined blocks providing common traffic engineering operations. The mechanics of constructing such a strategy can be viewed in three steps.

1. **SELECT:** Blocks providing the requisite device interfaces, signal processing, control computations, cycle phasings, or data collection features are selected and placed on the drawing area.

be represented internally as a data structure with local storage, inputs sockets, and output sockets. These data structures are different for each block type. For example, the DRUM block has 16 digital output sockets, but the OR block only has one digital output. To provide a structured method for interacting with the various data structures, a master list of blocks called the Block Table (Figure 7) maintains a list of all the symbolic block names and a code representing the class of blocks. For example, all OR blocks would have a class code of 11 and all D_UI blocks would have a class code of 19. This code is used by the software model to determine which table to look in to retrieve the data structure defining a block. For example, the table for OR blocks (Figure 7) would contain the data structures defining the EW_PRS and NS_PRS blocks and the COUNTER table (Figure 7) would contain the EB_CNT, WB_CNT, NB_CNT, and SB_CNT blocks.

Connections between blocks are very important for this model since they provide the mechanism for communication. The connection table (Figure 7) provides a list of all data connections and includes the following information:

- **Source socket** is a symbolic name identifying the source of a data connection.
- **Destination Socket** is a symbolic name identifying the destination of a data connection.
- **Socket Type** indicates what table to look in for the socket. For the data model shown in Figure 7, this could be a reference to the digital, analog, or text socket tables.
- **Socket State** indicates if the socket is an unconnected input, unconnected output, or connected (Figure 5).
- **Socket Index** is used to locate the particular socket in a socket table (Digital, Analog, or Text) identified by the socket type field.

In preceding sections, sockets have been conceptually diagrammed as tightly coupled with the block. However, in order to improve implementation efficiency, all sockets are stored outside the block and referenced via the connection and socket tables. There is a socket table for each possible data connection type. For example, digital states, analog values, or text messages are depicted in Figure 7. When a block is executed, it references its input socket indexes (Figure 7) and retrieves the appropriate information from the socket table. Similarly, after the computations have been performed, it uses the output socket indexes to update the respective output sockets.

Real Time Scheduling of Function Blocks. Since block processing is not instantaneous, the blocks must be scheduled such that all blocks have an opportunity to run often enough to meet their application requirements. One possible approach would be a round-robin scheduler. The problem with this type of scheduling is that when blocks are added or subtracted, the timing characteristics change. This kind of side effect is unacceptable, particularly if it is necessary to interact with a particular device or evaluate a traffic signal phase change at regular intervals. A more sophisticated approach would be to run all the blocks at their fastest required rate (a least common denominator approach). This technique would be adequate if sufficient CPU cycles were available for executing all blocks at the fastest required rate. However, in practice, only a few blocks require very frequent service (say 50Hz) and other blocks require service far less often (say .1Hz or 0.01Hz).

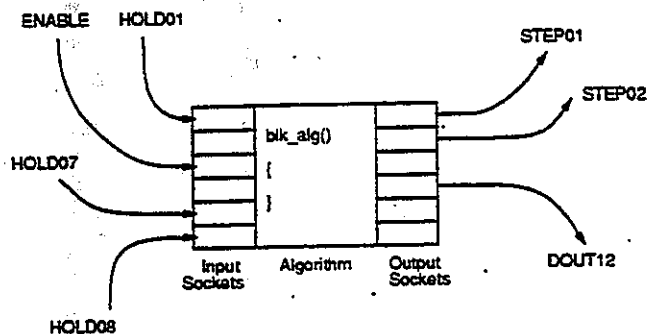


Figure 4: An illustration of function block components.

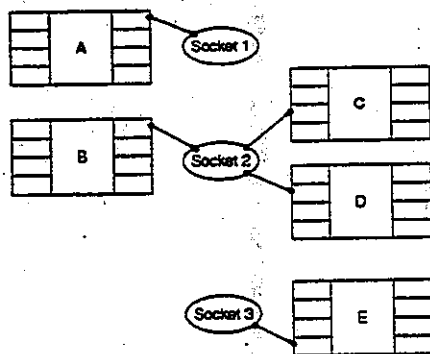


Figure 5: Example of valid socket connections

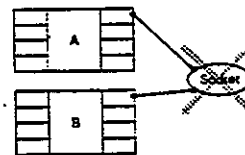


Figure 6: Example of invalid socket connection

5. Implementation of TCBLKS

Traffic engineers are likely to be most concerned with the block vocabulary, configuration concepts, and language structure of this traffic control software model. To round out the description of this function block model, a few important implementation concepts are addressed: i) internal data model for the function blocks, ii) real time scheduling, iii) capacity considerations, and iv) online user interfaces. Our purpose is not to formally define the model but to demonstrate an efficient real time implementation and to give the reader further insight into the software model. A more extended discussion appears in [Bullock 92b].

Consider the example strategy depicted in Figure 2. This strategy is composed of 32 blocks that describe what sensors should be read, which internal algorithms should be used (sequencers, counters, data collectors), and which actuators should be manipulated. Without regard to how often the blocks must be run, this strategy can be described as an topologically sorted list of blocks to be run {EB_PRS, WB_PRS, EW_PRS, NB_PRS, SB_PRS, NS_PRS, SEQNCR, SB_RED}. Each of these blocks must

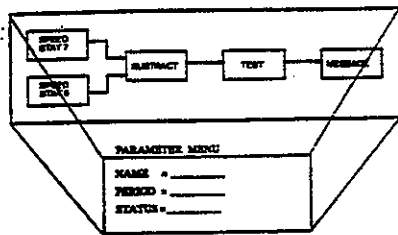


Figure 8: Block grouping illustration.

Task Table

Index	Period	Priority
-------	--------	----------

Group Table

Group	Task	Status
-------	------	--------

Figure 9: Scheduling tables.

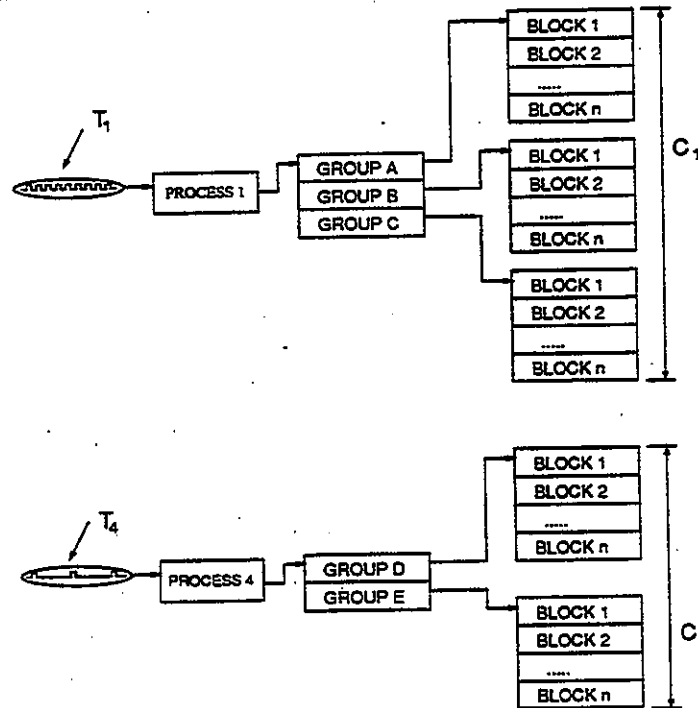


Figure 10: Processing of *group* and *block* structures by periodic task.

To facilitate the orderly startup and shutdown of a function block strategy, the software starts up in a single threaded model. It reads the function block strategy, creates all the necessary data structures for execution, initializes all I/O devices, runs all blocks once to initialize them, spawns periodic tasks, and commences the periodic execution depicted in Figure 10. The periodic tasks are created according to the period and priorities in the task table (Figure 9). Groups are assigned to these tasks according to the task field in the group table (Figure 9). When the software receives a signal to shutdown, it allows the periodic tasks to complete their current cycle (only if block processing was in progress before the shutdown signal was received), returns to single threaded operation, runs all blocks once (permits files to be closed, and I/O to be left in a safes state), and then terminates. The state diagram for this behavior is shown in Figure 11.

Block Table

Group	Block	Class
-------	-------	-------

Connection Table

Source	Destin	Type	Index
--------	--------	------	-------

OR, Class Code = 11

Group	Block	Local Storage	Input Sockets	Output Sockets
-------	-------	---------------	---------------	----------------

Digital Socket Table

Index	ddata	State
-------	-------	-------

D_UI, Class Code = 19

Group	Block	Local Storage	Input Sockets	Output Sockets
-------	-------	---------------	---------------	----------------

Analog Socket Table

Index	adata	State
-------	-------	-------

COUNTER, Class Code = 21

Group	Block	Local Storage	Input Sockets	Output Sockets
-------	-------	---------------	---------------	----------------

Text Socket Table

Index	tdata	State
-------	-------	-------

DRUM, Class Code = 26

Group	Block	Local Storage	Input Sockets	Output Sockets
-------	-------	---------------	---------------	----------------

A_COLL, Class Code = 301

Group	Block	Local Storage	Input Sockets	Output Sockets
-------	-------	---------------	---------------	----------------

Figure 7: Internal model for function blocks.

Due to the varying timing requirements for different portions of a block strategy, it is desirable to be able to assign a processing period to a group of blocks. To provide this capability and introduce a hierarchical level of abstraction, blocks can be grouped and assigned a name and periodic execution rate (Figure 8). Two additional internal tables are constructed to maintain this information, the Group Table, and the Task Table (Figure 9). An additional status field is used to turn the processing for an entire group of blocks on and off. From the users' perspective, a collection of groups assigned to periodic tasks constitute an application program (Figure 10). In the application shown in Figure 10, the blocks in Group A, Group B, and Group C would be run every T_1 seconds. Similarly, the blocks in Group D and Group E would be run every T_4 seconds. Within each of these groups, the blocks, their type, and their configuration define the semantics of the application program.

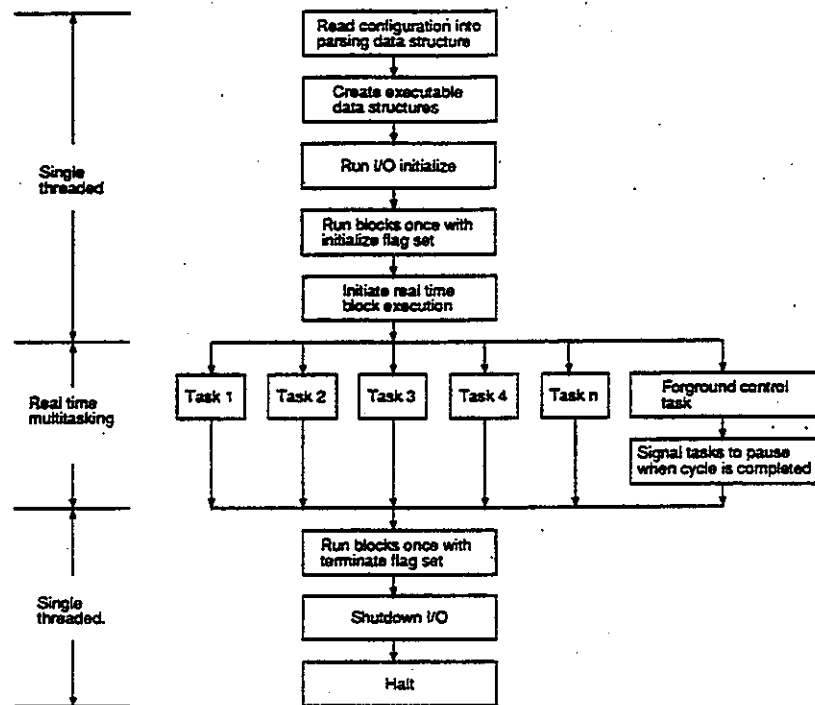


Figure 11: Software State Diagram.

Capacity considerations. The periodic tasks depicted in Figure 10 represent only one half of the software model. In practice, interactions with I/O devices such as serial ports, user interfaces, and disk drives have inherent time delays. To permit the processors to work on other duties, the periodic tasks do not directly interact with these devices. Instead, they communicate with asynchronous tasks using internal buffers. Conceptually, this software architecture look like that depicted in Figure 12. The periodic tasks that run the function blocks are shown on the left and the aperiodic tasks interacting with I/O devices are shown on the right. In examining Figure 12, a complex set of tasks is depicted. By inspection, it is not obvious whether or not the model can or will behave in a deterministic manner.

From Figure 10, we could intuitively say that for a set of six periodic tasks, with periods {0.1, 1.0, 2.0, 6.0, 30.0, and 60.0} (s), and corresponding execution times for processing all the blocks assigned to a task of {0.001, 0.001, 0.02, 0.04, 0.03, 2.0}, that the blocks probably would not overload the processor and the system would probably behave in a deterministic manner. In contrast, if the time required to process the blocks running ever 0.1 seconds was increased to say 0.20 seconds, it is clear the processor would be overloaded and the strategy would not run as expected. However, for task loading between these two extreme points, the feasibility of a given task set can not be determined by inspection. A formal verification model is required.

This particular processor scheduling problem has been studied for nearly three decades and can be

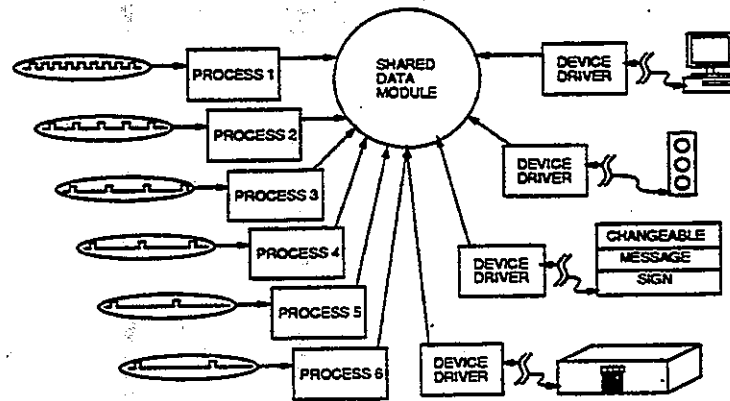


Figure 12: Task Interaction

addressed by the rate monotonic scheduling algorithm [Liu 73], with some minor extensions to accommodate aperiodic tasks that service asynchronous devices such as serial ports [Lehoczký 87]. A formal definition and proof of this scheduling algorithm can be found in Liu's seminal paper, but for our purposes it is sufficient to define a few descriptive measurements of the tasks running the function blocks and postulate two simple design rules.

- **Task Period**, denoted by T_i , indicates the period of the i^{th} task. These values determine the periodic intervals available for block execution and are typically {0.10, 0.50, 1.0, 30.0, 60.0, and 300.0} seconds, but can be changed by the traffic engineer to meet the requirements of a particular application.
- **Task Runtime** denoted by C_i , indicates the time required by the i^{th} task to process all the blocks assigned to it. These values can be computed by adding up the worst case processing time for each block assigned to a particular task.
- **Task Utilization**, denoted by U_i , indicates the percentage of CPU time consumed by the i^{th} task. These values are computed as the quotient C_i/T_i . The task utilizations values for the example described above are {0.010, 0.001, 0.01, 0.007, 0.001, 0.030}
- **Total Task Utilization**, denoted by U^* , indicates the percentage of CPU time consumed by all the periodic tasks. This is computed by summing up the task utilization U_i for each task. For the example described above, this value is 0.06.

For a task set conforming to the constraints of rate monotonic scheduling (which the function block model does), there are two different rules that have been proven which allow us to evaluate the feasibility of a given task set a priori:

1. A task set is guaranteed to be feasible (all task will meet their deadlines) if the sum of all task utilizations is less than 0.69 ($\ln 2$) [Liu 73]. This is a very simple and elegant check, but

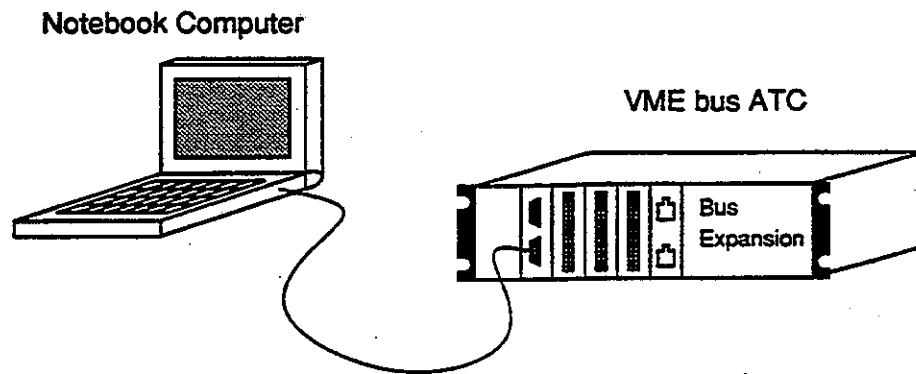


Figure 13: Use of a notebook computer to configure and monitor an Advanced Traffic Controller.

Interfacing with the controller in this fashion provides two important features. First, the client can symbolically reference any socket. So instead of the current practices on 170 controllers of looking at the word located at a particular hex offset, a symbolic name like "Main&4th;NB_CNT.AOUT" could be used to read the volume counter on the Northbound counter at Main and 4th. Second, the "State" field in the Connection table restricts the ability of an operator interface program to write to a socket to only those input sockets not connected to other blocks (Figure 5, Socket 3). Of course, any point could be read by an operator interface, but unpredictable operation would result if an operator was trying to change an output socket that was also being changed by a function block (Figure 5, Sockets 1 or 2).

6. Example Application for Freeway Ramp Metering

To provide a better understanding of control tasks which can be addressed, this section presents a more extensive example function block strategy.

Consider a freeway entrance that requires locally responsive ramp metering (Figure 14). Although this is a relatively common traffic engineering problem, nearly every agency has developed through an evolutionary approach their own control model to address local geometric and policy constraints. Since all the control software is written in assembly language, it is very difficult to share control concepts or explain exactly how a particular technique was implemented. The following example was developed to show how the function block model for the ATC could be applied to this problem. A few points regarding the strategy are worth noting. First, the semantics of the strategy can be identified by inspection of the computable block diagram. Second, a traffic engineer looking at this sketch can quite easily request new blocks that would be directly usable by the traffic engineer and not require any procedural programming on the part of the traffic engineer.

The control model selected for this example monitors the volume and occupancies from a set of loops adjacent to the ramp, computes a red interval based upon those volume and occupancies, and then selects

2. further research has shown that Liu's least upper bound on processor utilization is based upon a clustered task set that would not be encountered under typical circumstances. For example, it would be unlikely to have six task with the following clustered periods {0.98, 0.99, 1.00, 1.01, 1.02}. A closed form set of equations based only on the task periods T_1, \dots, T_n and task computation times C_1, \dots, C_n have been developed to provide a feasibility check for a given task set [Lehoczy 87]. Simulation studies have shown that for a task set that are not clustered, a processor utilization around 0.85 to 0.90 is usually feasible.

These rules provide a simple check permitting us to determine apriori if a function block strategy will meet all its deadlines, even under worst case conditions. If a strategy meets these requirements, then its behavior will be deterministic.

For redundancy purposes, it is generally accepted that one controller will be used per intersection. However, to give the reader a feel for the block capacity of a controller, consider the strategy shown in Figure 2. The 19 blocks in the top portion of the strategy are configured to execute at 2Hz, the CNTR blocks and their corresponding D_UI blocks that read the detector pulse inputs are configured to run at 10Hz, the D_UI blocks used to reset the CNTR blocks are configured to run at 1Hz, and the A_COLL block logging data is configured to run every 30 seconds. The periods for these tasks are {0.10, 0.50, 1.00, 30.00} and their corresponding task loadings have been computed to be {0.048, 0.011, 0.0008, 0.0006}. This means the strategy is only consuming approximately 6% of the available CPU cycles. If we only use the first design guideline ($U^* < 0.69$), up to 11 duplicates of the strategy shown in Figure 2 could run on one controller and operate with deterministic behavior. If we analyze the strategy using the exact closed form equations, up to 14 duplicates of the strategy shown in Figure 2 can run on one controller without overloading the processor.

The 10Hz task responsible for counting vehicles consumes the majority of the CPU cycles in this example. If these polled counters were omitted, up to 85 duplicates of the strategy shown in Figure 2 could be run. Work is currently under way to develop a specification for an intelligent I/O board that would maintain vehicle count and occupancy registers independently of the central processor. This would eliminate the need for blocks to poll I/O at relatively fast rates.

Online User Interfaces. In previous sections, we have described the user interface for configuring the block strategy. The user interface for instrumentation and monitoring, is also very important for development and diagnostic purposes. An interface such as the hex keypad and LED display found on the 170, or the alphanumeric display now being built into NEMA controllers could be used to interact with the ATC software. However, the function block model proposed in this paper provides a more intuitive method for interacting with the runtime control software. The basic concept for developing these "runtime user interfaces" is based upon a client/server model where the client is an operator interface program and the server is the function block processing program. Quite likely, the operator interface would be implemented on a notebook computer that could be plugged into a serial port on the ATC (Figure 13). The client operator interface would interact with a strategy via the connection table and the various sockets tables (Figure 7).

The blocks NB_PUL1 and NB_PUL2 monitor the pulse outputs of the adjacent loop detectors. The VPS1 and VPS2 block compute the rate, in vehicles per second, of the incoming pulses. The AvgVPS block has a gain of 0.5 assigned to each input, so each scan it outputs the average vehicles per seconds per lane passing over the detectors.

The remainder of the strategy shown in Figure 15 belongs to the group "RampCtl", which runs every 0.5 seconds (2 Hz). In this group, the average occupancy computed by the OCC_AVG block is filtered in the OCC_FILT block to remove "bumps" in the real time data. The filter used in this block is an approximate first order algorithm and the time constant can be set to a constant or changed dynamically via a connection into the block. The HzToVPH block converts the vehicles per second output by the AvgVPS block to vehicles per hour. That output of the HzToVPH is connected to the VOL_FILT block, which operates similar to the OCC_FILT.

The filtered occupancy is fed into the OCC_CTL, a lookup table that computes a recommended ramp meter red interval based upon the filtered occupancy. The operation of this block is depicted in Figure 16. The block is configured by specifying several x-y pairs defining a piecewise linear function. Every time the block runs, it computes an interpolated output using the lookup table and input value. The VOL_CTL block operates in a similar manner except instead of using a lookup table based upon *occupancy-red interval* pairs, the lookup table is defined by *volume-red interval* pairs. The outputs of the two lookup tables (OCC_CTL and VOL_CTL) are read by the SEL_RED block, which chooses the larger of two red intervals (most restrictive) for its output. The output of this block is the desired red interval for the ramp meter.

This red interval time is connected to two blocks, the signal drum sequencer called Meter and the test block called MinTst which shuts the meter down when the red interval is less then 7.2 seconds. The blocks named IntrLck and TurnOn connected to the drum sequencer provide the interlock that prevent the signal from shutting down during the red interval. The D_UI blocks named RED and GREEN are connected to the contacts controlling the lamps. The MESSG block names Status is used to display a message indicating if the ramp meter is on or off. The D_COLL block named MeterLog is used to log the times the meter turned on and turned off.

A traffic engineer might review this strategy and comment "... that is basically how we implement ramp metering, but". For example, many cities and town demand that they be permitted to dump vehicles on the freeway at an accelerated rate when a ramp begins to spills back into local streets. To illustrate the flexibility of this approach, a modified strategy providing this capability is shown in Figure 17. In this figure, the D_UI block named Q_PRS monitors a loop at the head of the ramp (Figure 14). To filter out transient pulses, this block is connected to a DLY_ON block named Debounce that delays turning on until the output of Q_PRS remains high for a specified duration (10s for this example). The output of the Debounce block is connected to the DLY_OFF block named QUEUE which remains on 60 seconds after its input, from the Debounce block, was last on. The output of this block is used by the strategy to determine if a queue it exists. The reason for providing the added delay before turning the queue detector off is to prevent "chatter" during congested periods.

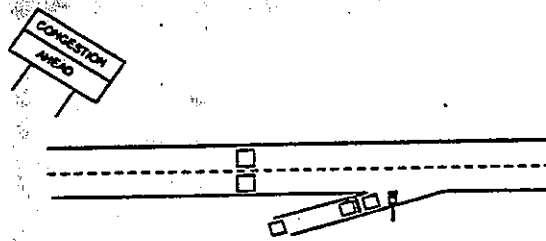


Figure 14: Ramp metering control problem.

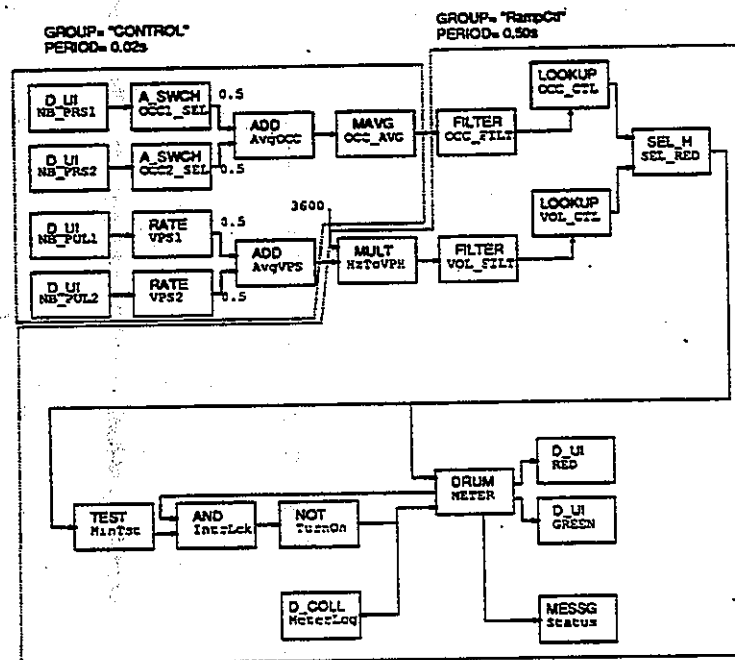


Figure 15: A responsive ramp meter control strategy.

the most restrictive of the two intervals. The actual implementation of this strategy is shown in Figure 15. In this strategy, the dotted line around the blocks in the upper left corner is used to denote the blocks belonging to the "SENSOR" group, which is used to monitor the detectors and estimate traffic parameters. The "SENSOR" group is specified to run every 0.02 seconds (50 Hz). The blocks NB_PRS1 and NB_PRS2 monitor the presence outputs of the adjacent loop detectors. The OCC1_SEL and OCC2_SEL blocks are switches that output either 0.0% or 100.0% if a vehicle is detected, or not detected, respectively. The AvgOcc block has a gain of 0.5 assigned to each input, so each scan it outputs the average occupancy across all lanes. This instantaneous occupancy is connected to the OCC_AVG to compute a moving average using a 25 point sample size.

7. Discussion

The software and examples described in this paper have been implemented and tested in real time under simulated conditions for applications such as signalized intersections, ramp metering, and communication with existing traffic control devices. The work was performed on the proposed Caltrans ATC platform configured with a 16 MHZ 68020 with 4 MB of RAM. Several observations have been noted during this work.

Assuming the current developments in the area of transportation controller hardware and software continue, it is reasonable to envision the emergence of a controller with the following characteristics:

1. A specification for an ATC computer platform that can be configured from a family of modular CPU's, I/O cards, network adapters, and serial ports depending upon the demands of a particular application.
2. A standard high level application development software model that can be used by traffic technicians and engineers to develop routine software such as signalized intersection control, supervision of multiple intersections, changeable message sign control, incident detection, ramp metering, data collection and so on. This software would be configured graphically on a notebook PC and down loaded into EEPROMS or flash memory in an ATC. The software running on the controller will be specifically designed to act as a server for developing client/server interfaces.

Neither the software or hardware portion of this vision are complete at this current time. Although the hardware is commercially available for such a controller, several issues remain including:

- Developing a list of "approved" modules and vendors. Since the VME bus only defines a hardware standard for integrating new modules, the software application interfaces to these "approved" modules must be standardized. For common modules such as digital I/O and serial I/O cards this standardization has already occurred. However, if specialized modules such as inductive loop interfaces are developed, the software interface must be explicitly considered.
- Defining standard mechanical and electrical field connectors for approved modules.
- Establishing OEM's to provide one stop ATC controller shopping.
- Commitment by state and local agencies to such a platform to develop a sufficient market for multiple vendors to compete in.

The software aspect of this controller is both the most important and most challenging aspect of this controller vision. If we do not devote sufficient resources toward developing a "standard software model," software development issues will severely impede modernization efforts. This type of development may be essential to achieve the benefits of proposed IVHS applications [Pline 92]. This paper proposed a software model that is commonly used in the commercial sector and could provide the following benefits:

- Basic software infrastructure for multiple applications. Since software would be configured by assembling function blocks and instrumented via socket tables, all controllers would have a uniform look and feel regardless of the application.

A safe red interval for dumping the vehicles is computed by the LOOKUP block named SAFE_RED. The lookup table used in this block differs from that used in VOL_CTL since it computes a minimum safe red interval instead of trying to maintain the optimal red interval. The SWITCH block named DUMP_RED is used to select between the minimum safe red interval (SAFE_RED) and the more restrictive optimal rate (SEL_RED). If a queue is detected, the output of DUMP_RED is the minimum safe red interval. If no queue is detected, the output of DUMP_RED is the optimal red interval. The remainder of the strategy mirrors that described for Figure 15.

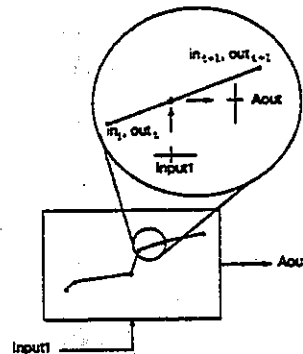


Figure 16: Operation of the lookup table block algorithm.

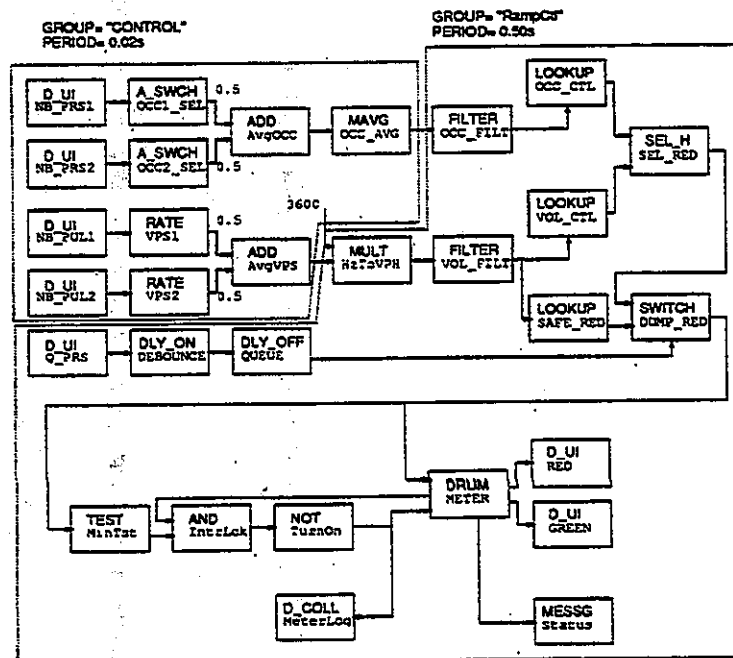


Figure 17: A responsive ramp meter control strategy with queue override.

- This model provides a much safer route for incorporating changes since the "application" is customized, rather than the real time programming model.
- Function block diagrams provide a mechanism for exchanging control concepts.
- Economically, it would not be feasible for states to each develop their own ATC software. To avoid becoming dependent upon a single software vendor, it is critical that the block definitions and interface protocols be precisely specified. This will permit competitive bidding of independent modules such as the graphical configuration tool, the real time kernel, real time user interface tools, and auxiliary function blocks.
- The structured nature of blocks provides a formal framework for traffic engineers to define and request new or revised blocks. As this software model grows, it is conceivable to imagine a TRB or ITE committee overseeing this development.

Acknowledgments

This work was supported in part by the California Department of Transportation under subcontract to the University of California Institute of Transportation Studies at Irvine. The views expressed by the authors do not necessarily reflect the individual views or policies of either the California Department of Transportation or the University of California.

References

- [Bullock 92a] Bullock, D. and C. Hendrickson. "Advanced Software Design and Standards for Traffic Signal Control," *Journal of Transportation Engineering, ASCE*, Vol. 118, No. 3, pp. 430-438, May 1992.
- [Bullock 92b] Bullock, D. and C. Hendrickson, *A Model for Roadway Traffic Control Software*, Technical Report, Carnegie Mellon University, December 1992.
- [Caltrans 92] California Department of Transportation, "Equipment Invoices," personal communication, 1992.
- [Chase 89] Chase, M.J. and Hensen, R.J., "Traffic Control Systems - Past, Present and Future," *Applications of Advanced Technologies in Transportation Engineering, ASCE*, pp. 257-262, February, 1989.
- [Garner 83] Garner, N.G., "OPAC: A Demand-Responsive Strategy for Traffic Signal Control," *Transportation Research Record*, No. 906, pp. 75-81, 1983.
- [Lehoczky 87] Lehoczky, J., L. Sha, and J. Strosnider, "Enhanced Aperiodic Responsiveness in Hard Real-Time Environments," *Real Time Systems Symposium*, IEEE Computing Society, pp. 261-270, 1987.
- [Liu 73] Liu, C.L. and Layland, J.W., "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal of the Association for Computing Machinery*, Vol. 20, No. 1, pp. 46-61, January 1973.
- [Pline 92] Pline, J.L., *Traffic Engineering Handbook, Fourth Edition*, Prentice Hall, 1992.
- [Quinlan 89] Quinlan, T., *Evaluation of Computer Hardware and High-Level Language Software for Field Traffic Control*, Technical Report, California Department of Transportation, Sacramento, CA, December 1989.
- [Rappaport 91] Rappaport, A. and Haleri, S., "The Computerless Computer Company," *Harvard Business Review*, Vol. 69, No. 4, pp. 69-80, July 1991.
- [Shaw 90] Shaw, M., *Prospects for an Engineering Discipline of Software*, Technical Report CMU-CS-90-165, Carnegie Mellon University, Pittsburgh, PA, September 1990.

APPENDIX C

SWIM Hardware Data Sheets and Installation Details

SWIM sensors include an inductive loop detector to indicate the presence of a vehicle and in turn initiate the data acquisition process, a series of eleven axle sensors to measure axle spacings and in turn produce vehicle class types, and two weigh pads to sense wheel loads and in turn provide vehicle weights. Data acquisition circuitry includes the ATC field controller, signal conditioning and sensor interface hardware, and peak-hold interrupt electronics. An operational description of the complete SWIM system is provided in the Operational Test Section 4.2.4, page 57. SWIM sensors, data acquisition hardware, and installation procedures are described in the following section.

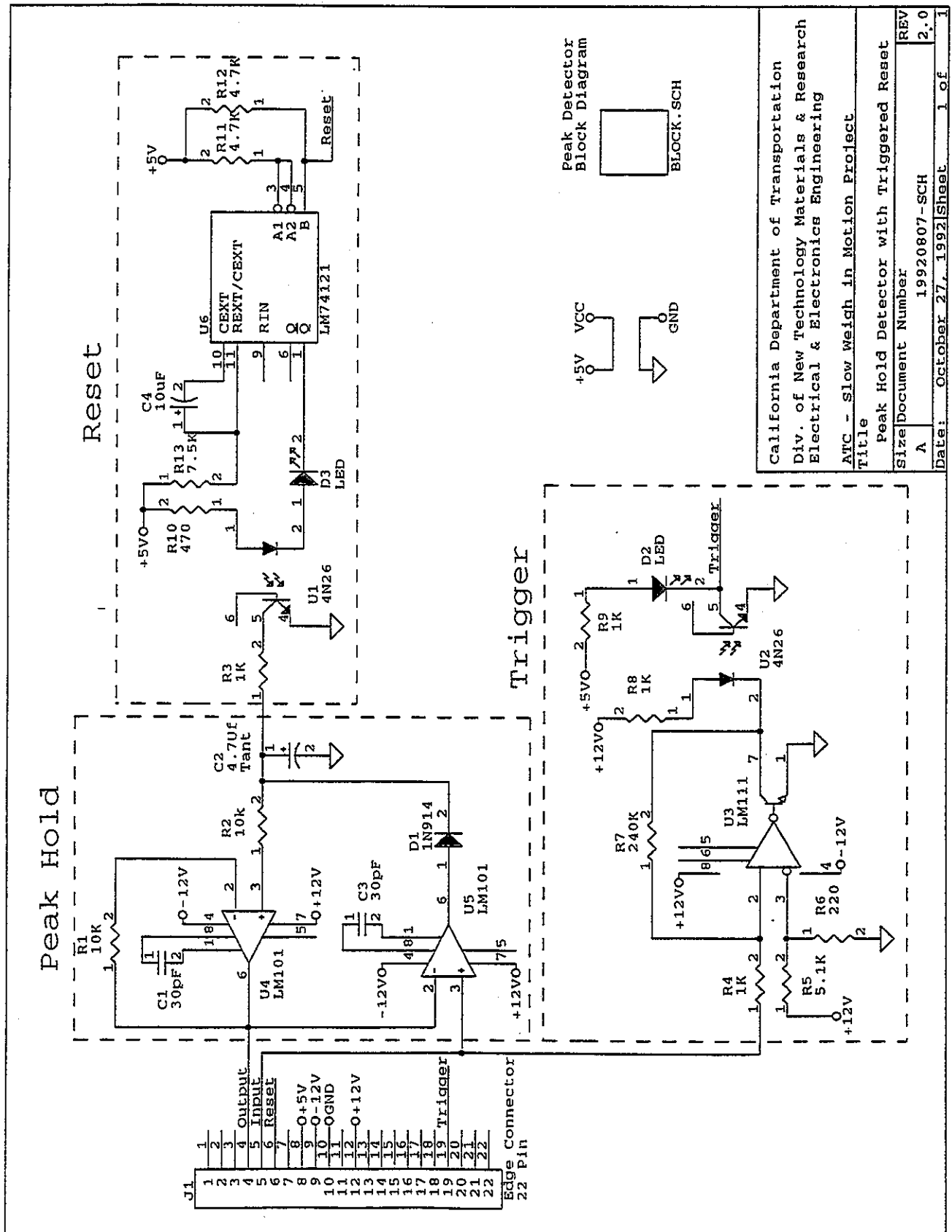
Peak Hold Circuitry

The peak hold circuit captures and holds the maximum amplitude of a positive voltage signal. As the wheel is applied to the weigh pad, it begins to output a rising positive signal (mV). The weigh pad amplifier circuitry amplifies the voltage to $1\text{ V} = 10,000\text{ lbs} = 10\text{ kip}$. As the input signal begins to rise the voltage is stored in the holding capacitor C2.

When the wheel is over the middle of the weigh pad, giving the maximum weight and voltage input to the peak hold circuit, the hold capacitor C2 is fully charged and the input signal begins to fall. When the input voltage drops 0.7 V below the maximum voltage held by C3, diode D1 turns on, holding the maximum voltage on the hold capacitor. The hold capacitor discharges at a rate of 1 mV/second, which is negligible in this situation. The next stages wait for the second trigger to occur.

The trigger circuit generates two triggers throughout the entire operation. The first trigger occurs as the input signal begins to rise above 0.45 V, sending an interrupt to the ATC controller. This interrupt is ignored by the computer and the peak hold circuit keeps charging. The second trigger occurs as the input signal begins to fall below 0.55 V, sending a second interrupt to the ATC controller. The controller reads the output voltage held by the peak hold circuit and in turn sends a signal to the peak hold reset circuit.

Finally, the controller sends a signal to the reset circuit to activate the one-shot. The one-shot clamps hold capacitor C2 to ground for approximately 50 ms. This allows the hold capacitor C2 to fully discharge before the next wheel is applied on the weigh plate. Now that the hold capacitor is fully discharged, the peak hold circuit is ready to capture the next wheel entering the system.



Specifications:

Power supply

±12V
+5V

Calculations for Trigger Circuit:

Switched Voltage

$$V_{SW} = \left(\frac{R_F + R_I}{R_F} \right) * V_R$$

Hysteresis Voltage

$$V_H = \left(\frac{R_I}{R_F} \right) * V_S$$

Reference Voltage

$$V_R = \left(\frac{R_P}{R_S + R_P} \right) * V_S$$

Results

$$\begin{aligned} V_{SW} &= .5V & R_I &= 1K\Omega \\ V_H &= 50mV & R_F &= 240K\Omega \\ V_S &= 12V & R_P &= 220\Omega \\ V_R &= .498V & R_S &= 5.1K\Omega \end{aligned}$$

Calculation for Reset Circuit:

One-Shot On Time

$$T = R_{13} * C_4 * \ln(2)$$

Results

$$T = 52ms \quad R = 7.5K\Omega \quad C = 10\mu F$$

Bill of Materials

Peak Hold Detector with Triggered Reset
19920807-SCH

Revised: October 1, 1992
Revision: 2.0

Item	Quantity	Reference	Part
1	2	C1,C3	30pF
2	1	C2	4.7Uf
3	1	C4	10uF
4	1	D1	1N914
5	2	D2,D3	LED
6	1	J1	Edge Connector
7	2	R2,R1	10k
8	4	R3,R4,R8,R9	1K
9	1	R5	5.1K
10	1	R6	220
11	1	R7	240K
12	1	R10	470
13	2	R11,R12	4.7K
14	1	R13	7.5K
15	2	U1,U2	4N26
16	1	U3	LM111
17	2	U4,U5	LM101
18	1	U6	LM74121

Installation Procedures

Installation of Replaceable Axle Sensor (10)

The axle sensor frame should be installed in a level portion of the roadway. It is strongly recommended that ruts be less than one half inch under a two foot straight edge.

The first step in the installation of the axle sensor is to saw cut the cavity in the pavement to accept the frame. Saw cut slot three inches deep by eight inches wide by one hundred inches long and six inches depth at end for a five inch electrical box. Start about inch past the lane edge. See Figure D1. Jack hammer concrete out of the cavity (Figure D3) and blow out all loose debris (Figure D4).

The next step is to bore eighteen anchor holes six inches deep by one inch diameter in the pavement, using the empty frame as a drilling guide. **DO NOT DRILL AT HOLES MARKED "A"**. With the frame centered in the slot, first bores should be about three inches from the end of the slot (Figures D5 and D6).

Install anchors in bar. Stainless anchors are supplied. Expandable anchors of any kind should not be used.

Electrical Conduit must be drained at its lower point.

Grease the four, twelve inch long threaded leveling legs and insert them in the holes marked "A". Level the frame flush with the road surface. This should be done without the sensor strip in the frame (Figures D7 and D8).

Remove the frame from the cavity in the pavement. Mix the epoxy as per instructions (Figure D9). Note: When using the Redipaks, you will not need to use all of the sand supplied with your package. The consistency should be fluid enough to flow under the Dynax frame. Approximately one half of the sand provided should be sufficient, as a rule of thumb. Fill all eighteen anchor holes and put a thin layer of epoxy in roadway cavity. Reinstall the frame as previously discussed, and re-level it (Figures D10 and D11).

Fill the remainder of the cavity with epoxy. Trowel flush with the roadway and allow to cure for three to six hours (depending on the temperature).

Any clean-up (example: epoxy on frame) can be done during the curing time, provided material is partially set.

Remove the leveling legs, unbolt and remove the clamping bars and install the axle sensor (Figure D12).

During removal of the sensor from the packaging, ensure that you do not unduly bend the sensor. **IF THE SENSOR IS EXCESSIVELY BENT, IRREVERSIBLE DAMAGE MAY OCCUR. DO NOT LIFT FROM THE CENTER OF THE SENSOR.** Instead, the sensor should be lifted by two people, with uniform support along the length of the sensor.

Installation of Weight Pads (11)

With oil chalk and marking paint, mark the exact installation point for frames and induction loops as well as for the drain water and cable pipes. Be sure to observe squareness to traffic direction.

Cut the marked frame border two and one quarter inches deep with the joint cutter; do not cross-cut over the corners, to avoid risk of future breaking. Cut the surface to be broken out for the frame into pieces. Do not cross-cut over the edges. Cut the excavation for the drain water pipe and the cable protection tube, four inches deep (Figures C1 and C2).

Excavate frame recess with electric or compressed air hammer (Figure C3). Cover inside of frames and bolts with duct tape to prevent the excess epoxy from adhering to the frame; leave anchor holes uncovered. This will save significant time in cleaning the frame and bolts of (necessary) overflow of Epoxy. Attach frame leveling aids to frame. Check depth with measuring template. Put frame in recess and mark breakout for cable outlet conduit and anchors.

Take frame out and drill anchor holes. Start vertical to about one half inch depth and then slowly turn to 45 degrees (Figure C13). Clean frame recess and anchor holes with compressed air gun (Figure C4). Surface must be dry and clean. Epoxy will only adhere to dry and clean rock, concrete or asphalt surface. Insert frame to test depth; (breakout to achieve sufficient depth).

Without adding hardener premix epoxy in sufficient quantity (three buckets per frame). Mix mortar on the dry side and build walls (Figure C14). The walls are only used to keep epoxy in place after pouring. The frame can be used for leveling. Remove excessive mortar and other loose debris in the bearing area and clean bearing area with compressed air and/or vacuum cleaner. This step is very important as any debris will reduce the bonding effect.

Install drain and cable conduit. Check bottom of frames and remove any debris which might adhere to the bituminous coat.

Close ends of pipes with paper plugs or duct tape.

Set the frame close to the recess and make the electrical connection with #8 bare copper wire underneath the frame at the provided one quarter inch bolts between the frame and the cable conduit. Before inserting the second and further frames, prepare the electrical connections.

Mix epoxy well with industrial mixer and add setting agent. Mix another two to three minutes especially at the edges (Figure C15). Processing time and setting time vary depending on the temperature. Do not use the epoxy remaining on the edges and the bottom of the container. This has not been mixed thoroughly with the setting agent and will not set properly. At temperatures below 50 degrees F., preheat the epoxy to about 80 degrees F. For installations in extreme temperatures, contact PAT for instructions.

Cast the support areas of the lowest frame including the anchor holes; level equal to the bank formed with the mortar (Figure C16). With a strong transverse slope, cast onto the higher side; wait, if need be, until the cast becomes a little consistent in order to avoid overflowing on the lower side. Apply Epoxy with a bristle brush to the vertical edges of the frame recess to obtain maximum bonding. If the bottom of the recess in the bearing area is not absolutely clean, also brush the bottom. **EPOXY WILL NOT ADHERE OTHERWISE AND CAN CAUSE EARLY LOOSENING OF THE FRAME.** Expect beginning of setting after approximately three to five minutes at high temperatures (above 70 degrees F.); ten minutes at medium temperatures (60 to 70 degrees F.); and fifteen minutes below 60 degrees F.

Immediately afterwards press in frame and insert anchors. Remove epoxy overflowing into the frame immediately. Make sure that the frame rests completely on the epoxy (Figure C17). In the event that pockets underneath the frame are suspected, immediately lift the frame and fill in with more epoxy.

For multiple frames, connect frames with the copper wire and insert the other frame parts one after another and make sure that there is no gap between the frame(s).

Mix cement mortar and close cavities between cable/drain water pipes and frame. Close the outgoing cable tube at the entry (to avoid incoming water up to one-third or one-half with cement mortar).

When the epoxy has set, after approximately 30 to 60 minutes, remove the leveling aids, remove duct tape, clean frame and insert the one quarter inch support rails in the frame (Figure C18).

Set the weighpad onto two wooden pieces with its upper side close by the frame and carefully push the connecting cable through the cable tube. Insert weighpad centrally into the frame with cable end first, then lower the other end down to one to two inches and let go (Figure C19). Grease the half inch bolts in the frame, the silicon rubber strip on the securing rails, and the slanted edge of the weighpad with anti-seize.

Insert securing rails and temporarily tighten with half inch fine nuts to approximately forty foot pounds to test liveliness. Verify that the weighpad corresponds to the roadway level over the entire width by means of a straight edge. If the weighpad is lower by more than 0.02 inch, adjust the shims accordingly and check again (allowable tolerance is ± 0.02 inch). Insert shims to compensate for the difference in height. The shims are available in thickness of 1/32 inch and 1/16 inch. Exact level is important in the wheel track.

Put support rail over shims, insert weighpad centrally again and attach securing rails with nuts and washers provided. Tighten nuts with a torque wrench to approximately 90 foot pounds. Fill open joints at the border and the cable tubes with 3M loop sealant until they are flush with the securing rails and the roadway level.

Verify installation height again.

Seal the joints between the weighpads and the frame, as well as between the securing rails and the frame and the recessed nut holes with silicon sealer, in particular at the cable connection.

Open the roadway to traffic after 2 to 5 hours (Figures C20 and C21).

APPENDIX D
ATC SWIM Prototype Software

Several routines service axle sensors and weigh pads that are collecting axle and wheel load data. A "compute" routine processes acquired data, producing axle counts, axle loads, axle group loads, gross vehicle weights, speeds, center-to-center axle spacings, axle group spacings, vehicle classes (one of fifteen based on axle spacings and weights), site identification, date and time stamps, sequential vehicle record numbers, California Vehicle Code (CVC) weight violations (including Bridge Law violations), and a display of summary data. Numerous functions within the IRQ DIN Module service sensors and initialize, open/close, activate/deactivate, and login/logout devices. Appendix D provides complete source code listings.

Neither the State of California, nor the United States Government, or any officer or employee thereof is responsible for any damage or liability occurring by reason of use of the ATC/WIM programs.

Program	Function
---------	----------

irq_din.c	source code for vin i/o control
-----------	---------------------------------

compute.c	source code for calculation of speed, axle spacing and weight
-----------	---

trkrec.h	header file for compute.c
----------	---------------------------

irq_din.h	header file for irq_din.c
-----------	---------------------------

makefile	compiles source code, links and loads irq_din program
----------	---

makepron	makefile for irq_din pron program
----------	-----------------------------------

i	initializes devices
---	---------------------

pron_vin	batch file for pron burning
----------	-----------------------------

swin_sim.c	truck simulation program for laboratory testing
------------	---

NOTE: Flow diagrams for irq_din.c were presented earlier in the body of the report, figures 33 through 45; simulation flow diagrams are found in the swin_sim.c section of this appendix.

CALIFORNIA DEPARTMENT OF TRANSPORTATION

== SLOW SPEED WEIGH-IN-MOTION SYSTEM SOFTWARE ==

===== DEVELOPMENT ENVIRONMENT =====

Development System: VM20 (Pep Modular Computers Inc.)
Operating System : OS-9 (Microvare)

=====

Hardware requirements: 2 VMOD Cards.

3 PB-DIN Piggybacks.

1 PB-DIO Piggyback.

1 VDAD Card.

*****/

=====
Module name: irq_din.c
=====

==SYSTEM CONFIGURATION ==

VMOD1 (Piggyback A)

H1 = Axle Sensor 0 (Active Lo)	Device: dix0a
H2 = Axle Sensor 1 (Active Lo)	dix0a
H3 = Axle Sensor 2 (Active Lo)	Device: dix0b
H4 = Axle Sensor 3 (Active Lo)	dix0b

VMOD1 (Piggyback B)

H1 = Axle Sensor 4 (Active Lo)	Device: dix1a
H2 = Axle Sensor 5 (Active Lo)	dix1a
H3 = Axle Sensor 6 (Active Lo)	Device: dix1b
H4 = Axle Sensor 7 (Active Lo)	dix1b

VMOD2 (Piggyback A)

H1 = Axle Sensor 8 (Active Lo)	Device: dix2a
H2 = Axle Sensor 9 (Active Lo)	dix2a
H3 = Axle Sensor10 (Active Lo)	Device: dix2b
H4 = VDAD Trigger (Active Lo)	dix2b

VMOD2 (Piggyback B)

H1 = Loop Sensor (Active - & +)	Device: dix3a
H4 = VDAD Reset (Active Hi -> Low)	Device: dix3b

=====*/

```
#include <errno.h>
#include <stdio.h>
#include <xodes.h>
#include <sgstat.h>
#include <strings.h>
#include "/dd/BSP/DEFS/vdaddefs.h"
#include "irq_din.h"
```

```
extern float AXLE_SPACE[9][4];
extern int WEIGHT[9],
        WEIGHT_LEFT[9],
        WEIGHT_RIGHT[9];
```

* Intercept routine: signal handler */

```
sig_hdlr(signal)
nt signal;
```

```
sigmask(1);
```



```

sig_in = signal;
}

/* Main program starts here */
main()
{
    int t, j,Ax_Sensor,bx;

    fpi = fopen("/r0/DATA.txt","w");          /* File for truck data */
    /***** This section may be deleted *****/
    for(j=0;j<25;j++) printf("\n");
    for(j=0;j<10;j++) printf("\n");
    printf("\t\t\t\t\t*** System is running ***\r\n\n\n");
    puts("            --- Hit CTRL-Z to Exit program ---");
    for(j=0;j<10;j++) printf("\n");
    /*****/

    /* Install intercept routine */

    intercept(sig_hndlr);

    /* Open devices for all input signals (axle sensors and weight pads) */

    open_devices();

    /* Assign signal numbers for all sensors, loop and weigh pad inputs */

    V1a_sig_H1 = 0x60; V1a_sig_H2 = 0x61; V1a_sig_H3 = 0x62; V1a_sig_H4 = 0x63;
    /* Interrupt trigger for axle sensor S0; S1; S2; S3; */
    V1b_sig_H1 = 0x70; V1b_sig_H2 = 0x71; V1b_sig_H3 = 0x72; V1b_sig_H4 = 0x73;
    /* Interrupt trigger for axle sensor S4; S5; S6; S7; */
    V2a_sig_H1 = 0x80; V2a_sig_H2 = 0x81; V2a_sig_H3 = 0x82; V2a_sig_H4 = 0x83;
    /* Interrupt trigger for axle sensor S8; S9; S10; weigh pad #1 peak hold; */
    V2b_sig_H1 = 0x90; V2b_sig_H3 = 0x91;
    /* Interrupt trigger for loop detector; weigh pad #2 peak hold; */
    control = 0x00;                /* Control byte of VDAD: set for software trigger */

    Init_S7_S10();                 /* Initialize variables of sensor 7 to 10 */
    activate_sensors();             /* Activate all axle sensors */

    i = 0;
    lenId = 0;                     /* Initialize truck Id number only once */
    while (i==0)

        sleep(0);                  /* wait for signal */

    switch(sig_in)
    {
        /*----- Axle Sensor 0 -----*/
        case(0x60): if(StartTruck)      /* Vehicle enters the loop */
            {
                Init_Var();              /*Initialize all variables */
                StartTruck = False;
                Prev_Total_Ax_Count = 0;
            }
        Ax_Sensor = 0;               /* Variable uses for indexing */
        Count[Ax_Sensor] += 1;       /* Increment sensor 0 count */
        if(Count[Ax_Sensor] > 1)     /* Start timer if end axle hits S0 */

```

```

if(Timer1_Status == AVAIL)
{
/* TOBS - Timer Occupied By Sensor is an array used to indicate which
sensor uses which timer. Two timers are required.
In this case, sensor 0 used timer 1 */
TOBS[Ax_Sensor] = TIMER1;
Timer1_Status = BUSY;
Setup_Timer1();
}
else
{
if(Timer2_Status == AVAIL)
{
TOBS[Ax_Sensor] = TIMER2;
Timer2_Status = BUSY;
Setup_Timer2();
}
}
}
break;

/*----- Axle Sensor 1 -----*/
case(0x61): Count[1] += 1; /* Increment sensor 1 hit count */
Service_Sensor1_5(1);
break;

/*----- Axle Sensor 2 -----*/
case(0x62): Count[2] += 1; /* Increment sensor 2 hit count */
Service_Sensor1_5(2);
break;

/*----- Axle Sensor 3 -----*/
case(0x63): Count[3] += 1; /* Increment sensor 3 hit count */
Service_Sensor1_5(3);
break;

/*----- Axle Sensor 4 -----*/
case(0x70): Count[4] += 1; /* Increment sensor 4 hit count */
Service_Sensor1_5(4);
break;

/*----- Axle Sensor 5 -----*/
case(0x71): Count[5] += 1; /* Increment sensor 5 hit count */
Service_Sensor1_5(5);
break;

/*----- Axle Sensor 6 -----*/
case(0x72): Count[6] += 1; /* Increment sensor 6 hit count */
if(Count[6] == Total_Ax_Count) /* Did last axle of the vehicle just hit sensor 6? */
{
EndRec = True; /* Set EndRec flag */
/*++ Sum up the left and right wheel weights. ++*/
for(j = 0; j < Total_Ax_Count; j++)
WEIGHT[j] = WEIGHT_LEFT[j] + WEIGHT_RIGHT[j];
Get_Ax_Spacings(); /* Call compute module to calculate and display results */
Prev_Total_Ax_Count = Total_Ax_Count;
/* Make 2nd copy of Total_Ax_Count */
/* Reset sensor 7-10 count */
Count[7] = 0;
Count[8] = 0;
Count[9] = 0;
Count[10] = 0;
Prev_Total_Ax_Count = 1; /* To avoid false latch with Count[7-10] */
truck_gone = True; /* Set flag truck_gone */
}
else
Service_Sensors_10(6);

```

```

break;
/*----- Axle Sensor 7 -----*/
case(0x71): Count[7] += 1; /* Increment sensor 7 hit count */
/* Check condition to catch vehicle with long rear axle to end of
   vehicle spacing. This condition is true when the last axle hits
   sensor 6 but the loop detector still register the presence of
   the same vehicle. */
if((Count[7] == Total_Ax_Count) && (!truck_gone))
{
    EndRec = true;
    /* Sum up the left and right wheel weights */
    for(j = 0; j < Total_Ax_Count; j++)
        WEIGHT[j] = WEIGHT_LEFT[j] + WEIGHT_RIGHT[j];
    printf("%d\t %d\n", VehId, Total_Ax_Count);
    Get_Ax_Spacings();
    Prev_Total_Ax_Count = Total_Ax_Count;
    Count[7] = 0;
    TOBS[7] = IDLE;
}
else /* "else" means normal vehicle exit */
{
    if(Count[7] == Prev_Total_Ax_Count) /* if last axle of vehicle */
    {
        Count[7] = 0;
        Count[8] = 0;
        Count[9] = 0;
        Count[10] = 0;
        Prev_Total_Ax_Count = 1;
        TOBS[7] = IDLE;
    }
    else /* need to locate following axle */
        if(Count[7] < Count[0])
            Service_Sensor6_10(7);
}
break;
/*----- Axle Sensor 8 -----*/
case(0x80): Count[8] += 1; /* Increment sensor 8 hit count */
if(Count[8] == Prev_Total_Ax_Count) /* Last axle of vehicle just hit sensor 8 */
{
    Count[8] = 0;
    Count[9] = 0;
    Count[10] = 0;
    Prev_Total_Ax_Count = 1;
    TOBS[8] = IDLE;
}
else
    if(Count[8] < Count[0])
        Service_Sensor6_10(8);
break;
/*----- Axle Sensor 9 -----*/
case(0x81): Count[9] += 1; /* Increment sensor 9 hit count */
if(Count[9] == Prev_Total_Ax_Count)
{
    Count[9] = 0;
    Count[10] = 0;
    Prev_Total_Ax_Count = 1;
    TOBS[9] = IDLE;
}

```

```

else
    if(Count[9] < Count[0])
        Service_Sensor5_10(9);
    break;
/*----- Axle Sensor 10 -----*/
case(0x32): Count[10] += 1; /* Increment sensor 10 hit count */
    if(Count[10] == Prev_Total_Ax_Count)
    {
        Count[10] = 0;
        Prev_Total_Ax_Count = 1;
        TOBS[10] = IDLE;
    }
    else
        if(Count[10] < Count[0])
            Service_Sensor5_10(10);
    break;
/*----- Input Trigger Signal from Peak Holder Circuitry -----*/
case(0x33):
    if(Temp_Count_S1 == Count[1] - 1) /* Handle weight pad # 1 */
    {
        Read_Weight(Temp_Count_S1,0); /* Read wt. pad #1 1/D conversion */
        Temp_Count_S1 = Count[1];
    }
    break;
case(0x31): /* Handle weight pad # 2 */
    if(Temp_Count_S3 == Count[3] - 1)
    {
        Read_Weight(Temp_Count_S3,1); /* Read wt. pad #2 1/D conversion */
        Temp_Count_S3 = Count[3];
    }
    break;
/*----- Vehicle Loop Detector -----*/
case(0x90): ++EndTruck_Flag;
    if(EndTruck_Flag > 2) /* Falling edge of loop signal */
    {
        StartTruck = True; /* Vehicle just entered system */
        ++VehId; /* Increment vehicle count */

        /* Set vehicle detector sensor to sense rising edge */

        if (_ss_PBDIY_sns(din_num7,1) == -1)
            exit(_errmsg(errno, " can't set sense H1 & H2 on %s\n",din_num7));
    }
    else
    {
        Total_Ax_Count = Count[0];
        EndTruck_Flag = 0;

        /* Reset loop detector digital input sensor to sense signal falling edge */
        if (_ss_PBDIY_sns(din_num7,0) == -1)
            exit(_errmsg(errno, " can't set sense H1 & H2 on %s\n",din_num7));
    }
    break;
/*----- Use other signals to terminate program such as CTRL-C -----*/
default: i = 1;
}

Activate_Sensors();
Use_Devices();

```

```

}                                     /* End of main program */

/*****
delay(unit)
{
int x;

for(x=0;x<unit;x++);
}
*****/
Close_Devices()
{
/***** Close input devices for all axle sensors *****/

printf("\n");
if( close(din_nua1) == -1 )
    exit(_errasg(errno," can't close %s\n",din_nua1));
if( close(din_nua2) == -1 )
    exit(_errasg(errno," can't close %s\n",din_nua2));
if( close(din_nua3) == -1 )
    exit(_errasg(errno," can't close %s\n",din_nua3));
if( close(din_nua4) == -1 )
    exit(_errasg(errno," can't close %s\n",din_nua4));
if( close(din_nua5) == -1 )
    exit(_errasg(errno," can't close %s\n",din_nua5));
if( close(din_nua6) == -1 )
    exit(_errasg(errno," can't close %s\n",din_nua6));
if( close(din_nua7) == -1 )
    exit(_errasg(errno," can't close %s\n",din_nua7));
if( close(din_nua8) == -1 )
    exit(_errasg(errno," can't close %s\n",din_nua8));

/***** Close input device for weight pads *****/

VDAD_Logout();
if( close(pdad1) == -1 )
    exit(_errasg(errno," can't close %s\n",dadlin));
printf("\n");
}
*****/
VDAD_Logout()
{
if( _ss_VDADi(SS_Logout,pdad1) == -1 )
    exit(_errasg(errno," can't logout %s\n",dadlin));
}

/*****
Routine to open all the device drivers of VMOD and VDAD cards.
*****/
Open_Devices()
{
/* Assigned devices path names to variables (Axle sensors) */

strcpy(din_nua1,"/dix0a");
strcpy(din_nua2,"/dix0b");
strcpy(din_nua3,"/dix1a");
strcpy(din_nua4,"/dix1b");
strcpy(din_nua5,"/dix2a");

```

```
strcpy(din_nam6,"/dix2b");
strcpy(din_nam7,"/dix3a");
strcpy(din_nam8,"/dix1b");
```

```
/* Assigned device path name to variable (Weight pads) */
```

```
strcpy(dadlin,"/dadlin");
```

```
/****** Open all devices for axle sensors *****/
```

```
if ((din_num1 = open(din_nam1,S_IRRD)) == -1)
    exit(_errnsq(errno," can't open %s\n",din_nam1));
if ((din_num2 = open(din_nam2,S_IRRD)) == -1)
    exit(_errnsq(errno," can't open %s\n",din_nam2));
if ((din_num3 = open(din_nam3,S_IRRD)) == -1)
    exit(_errnsq(errno," can't open %s\n",din_nam3));
if ((din_num4 = open(din_nam4,S_IRRD)) == -1)
    exit(_errnsq(errno," can't open %s\n",din_nam4));
if ((din_num5 = open(din_nam5,S_IRRD)) == -1)
    exit(_errnsq(errno," can't open %s\n",din_nam5));
if ((din_num6 = open(din_nam6,S_IRRD)) == -1)
    exit(_errnsq(errno," can't open %s\n",din_nam6));
if ((din_num7 = open(din_nam7,S_IRRD)) == -1)
    exit(_errnsq(errno," can't open %s\n",din_nam7));
if ((din_num8 = open(din_nam8,S_IRRD)) == -1)
    exit(_errnsq(errno," can't open %s\n",din_nam8));
```

```
/****** Set VDAD reset line to low means stop resetting *****/
```

```
if (_ss_PBDIX_ctl(din_num7,1) == -1)
    exit(_errnsq(errno," can't assert H4 on %s\n",din_nam7));
if (_ss_PBDIX_ctl(din_num8,1) == -1)
    exit(_errnsq(errno," can't assert H4 on %s\n",din_nam8));
```

```
/****** Open device for weight pads *****/
```

```
if ((pdad1 = open(dadlin,S_IRRD)) == -1)
    exit(_errnsq(errno," can't open %s\n",dadlin));
VDAD_Login();
```

```
/******
```

```
VDAD_Login()
```

```
{
    if(_ss_VDADi(SS_Login,pdad1,control,0) == -1)
    {
        printf("error %d: can't login %s\n",errno,dadlin);
        exit(errno);
    }
}
```

```
/******
```

```
Activate_Sensors()
```

```
/****** Enable interrupt request for VMOD 1 handshake pins - Piggy back 1 *****/
```

```
if (_ss_PBDIX_sns(din_num1,0) == -1)
    exit(_errnsq(errno," can't set sense H1 & H2 on %s\n",din_nam1));
```

```

if (_ss_PBDIX_en(din_num1,1,V1a_sig_H1) == -1)
    exit(_errmsg(errno," can't enable H1 IRQ on %s\n",din_num1));
if (_ss_PBDIX_en(din_num1,2,V1a_sig_H2) == -1)
    exit(_errmsg(errno," can't enable H2 IRQ on %s\n",din_num1));
if (_ss_PBDIX_sns(din_num2,0) == -1)
    exit(_errmsg(errno," can't set sense H3 & H4 on %s\n",din_num2));
if (_ss_PBDIX_en(din_num2,1,V1a_sig_H3) == -1)
    exit(_errmsg(errno," can't enable H3 IRQ on %s\n",din_num2));
if (_ss_PBDIX_en(din_num2,2,V1a_sig_H4) == -1)
    exit(_errmsg(errno," can't enable H4 IRQ on %s\n",din_num2));

/**** Enable interrupt request for VMOD 1 handshake pins - Piggy back 3 *****/

if (_ss_PBDIX_sns(din_num3,0) == -1)
    exit(_errmsg(errno," can't set sense H1 & H2 on %s\n",din_num3));
if (_ss_PBDIX_en(din_num3,1,V1b_sig_H1) == -1)
    exit(_errmsg(errno," can't enable H1 IRQ on %s\n",din_num3));
if (_ss_PBDIX_en(din_num3,2,V1b_sig_H2) == -1)
    exit(_errmsg(errno," can't enable H2 IRQ on %s\n",din_num3));
if (_ss_PBDIX_sns(din_num4,0) == -1)
    exit(_errmsg(errno," can't set sense H3 & H4 on %s\n",din_num4));
if (_ss_PBDIX_en(din_num4,1,V1b_sig_H3) == -1)
    exit(_errmsg(errno," can't enable H3 IRQ on %s\n",din_num4));
if (_ss_PBDIX_en(din_num4,2,V1b_sig_H4) == -1)
    exit(_errmsg(errno," can't enable H4 IRQ on %s\n",din_num4));

/**** Enable interrupt request for VMOD 2 handshake pins - Piggy back 4 *****/

if (_ss_PBDIX_sns(din_num5,0) == -1)
    exit(_errmsg(errno," can't set sense H1 & H2 on %s\n",din_num5));
if (_ss_PBDIX_en(din_num5,1,V2a_sig_H1) == -1)
    exit(_errmsg(errno," can't enable H1 IRQ on %s\n",din_num5));
if (_ss_PBDIX_en(din_num5,2,V2a_sig_H2) == -1)
    exit(_errmsg(errno," can't enable H2 IRQ on %s\n",din_num5));
if (_ss_PBDIX_sns(din_num6,0) == -1) /* May need to check sense here */
    exit(_errmsg(errno," can't set sense H3 & H4 on %s\n",din_num6));
if (_ss_PBDIX_en(din_num6,1,V2a_sig_H3) == -1)
    exit(_errmsg(errno," can't enable H3 IRQ on %s\n",din_num6));
if (_ss_PBDIX_en(din_num6,2,V2a_sig_H4) == -1)
    exit(_errmsg(errno," can't enable H4 IRQ on %s\n",din_num6));

/**** Enable interrupt for Loop Sensor ****/

if (_ss_PBDIX_sns(din_num7,0) == -1)
    exit(_errmsg(errno," can't set sense H1 & H2 on %s\n",din_num7));
if (_ss_PBDIX_en(din_num7,1,V2b_sig_H1) == -1)
    exit(_errmsg(errno," can't enable H1 IRQ on %s\n",din_num7));

/**** Enable interrupt for VDAD Trigger ****/

if (_ss_PBDIX_sns(din_num8,0) == -1)
    exit(_errmsg(errno," can't set sense H3 & H4 on %s\n",din_num8));
if (_ss_PBDIX_en(din_num8,1,V2b_sig_H3) == -1)
    exit(_errmsg(errno," can't enable H3 IRQ on %s\n",din_num8));

}

/*****
Deactivate_Sensors()
*****/

```

```

/***** Disable IRQ on H1, H2, H3, H4 for VMOD 1 - Piggyback A *****/

```

```

if (_ss_PBDIX_dis(din_num1,1) == -1)
    exit(_errasg(errno," can't disable H1 IRQ on %s\n",din_num1));
if (_ss_PBDIX_dis(din_num1,2) == -1)
    exit(_errasg(errno," can't disable H2 IRQ on %s\n",din_num1));
if (_ss_PBDIX_dis(din_num2,1) == -1)
    exit(_errasg(errno," can't disable H3 IRQ on %s\n",din_num2));
if (_ss_PBDIX_dis(din_num2,2) == -1)
    exit(_errasg(errno," can't disable H4 IRQ on %s\n",din_num2));

```

```

/**** Disable IRQ on H1, H2, H3, H4 for VMOD 1 - Piggyback B *****/

```

```

if (_ss_PBDIX_dis(din_num3,1) == -1)
    exit(_errasg(errno," can't disable H1 IRQ on %s\n",din_num3));
if (_ss_PBDIX_dis(din_num3,2) == -1)
    exit(_errasg(errno," can't disable H2 IRQ on %s\n",din_num3));
if (_ss_PBDIX_dis(din_num4,1) == -1)
    exit(_errasg(errno," can't disable H3 IRQ on %s\n",din_num4));
if (_ss_PBDIX_dis(din_num4,2) == -1)
    exit(_errasg(errno," can't disable H4 IRQ on %s\n",din_num4));

```

```

/***** Disable IRQ on H1, H2, H3, H4 for VMOD 2 - Piggyback A *****/

```

```

if (_ss_PBDIX_dis(din_num5,1) == -1)
    exit(_errasg(errno," can't disable H1 IRQ on %s\n",din_num5));
if (_ss_PBDIX_dis(din_num5,2) == -1)
    exit(_errasg(errno," can't disable H2 IRQ on %s\n",din_num5));
if (_ss_PBDIX_dis(din_num6,1) == -1)
    exit(_errasg(errno," can't disable H3 IRQ on %s\n",din_num6));
if (_ss_PBDIX_dis(din_num6,2) == -1)
    exit(_errasg(errno," can't disable H4 IRQ on %s\n",din_num6));

```

```

/**** Set VDAD reset line back to high ****/

```

```

if (_ss_PBDIX_cntl(din_num7,0) == -1)
    exit(_errasg(errno," can't assert H2 on %s\n",din_num7));

```

```

if (_ss_PBDIX_cntl(din_num8,0) == -1)
    exit(_errasg(errno," can't assert H4 on %s\n",din_num8));

```

```

/**** Disable vehicle Loop Sensor ****/

```

```

if (_ss_PBDIX_dis(din_num7,1) == -1)
    exit(_errasg(errno," can't disable H1 IRQ on %s\n",din_num7));
if (_ss_PBDIX_dis(din_num8,1) == -1)
    exit(_errasg(errno," can't disable H3 IRQ on %s\n",din_num8));
}

```

```

/*****
Routine to initialize all variables after reading a complete vehicle record
*****/

```

```

Init_Var()

```

```

{
    int index , k;

```

```

/** Global Variables Reinitialization after one truck **/

```

```

for(index = 0; index < 7; ++index)
{
    Count[index] = 0;

```



```

TCBS[index] = IDLE;
}

for(index = 0; index < 9; ++index)
{
    for(k = 0; k < 4; ++k)
        AXLE_SPACE[index][k] = 0.0;
    WEIGHT[index] = 0;
    WEIGHT_LEFT[index] = 0;
    WEIGHT_RIGHT[index] = 0;
}
Timer1_Status = AVAIL;
Timer2_Status = AVAIL;
Time1 = 0;
Time2 = 0;
Axle_Dist_Cnt = 0;
Total_Ax_Count = 0;
dist = 0;
EndRec = False;
truck_gone = False;
Temp_Count_S1 = 0;
Temp_Count_S3 = 0;
} /* End Init_Var */
/*****
Routine to initialize sensors 7 - 10 variables
*****/
Init_S7_S10()
{
    int j;

    EndTruck_Flag = 0;
    Prev_Total_Ax_Count = 0;

    /*** Initialize variables for sensor 7 to sensor 10 ***/
    for(j = 7; j < 11; j++)
    {
        Count[j] = 0;
        TCBS[j] = IDLE;
    }
}
/*****
Setup_Timer1()
*****/
{
    /*** Initializing PB1-DIN Timer of VH001 *****/

    TCR1 = 0x90; /* Do not use interrupt */
    CPRH1 = 0xFF; /* Load timer1 high byte count value */
    CPM1 = 0xFF; /* Load timer1 middle byte count value */
    CPL1 = 0xFF; /* Load timer1 low byte count value */
    TSR1 = 0x01; /* Write 1 to timer status register to clear */
    TCR1 = 0x81; /* Enable timer1 */
}
/*****
Setup_Timer2()
*****/
{
    /*** Initializing PB1-DIN Timer of VH002 *****/

    TCR2 = 0x90;
    CPRH2 = 0xFF;
    CPM2 = 0xFF;

```

```

CPRL2 = 0xFF;
PSR2 = 0x01;
PC32 = 0x31;
}
/*****
Routine to handle timers when an axle hits sensors 1 to 5
*****/
Service_Sensor1_5(Axle_Sensor)
int Axle_Sensor;
{
    if((Count[Axle_Sensor] > 1)&&(Count[Axle_Sensor] > Axle_Dist_Cnt+1))
    {
        switch(TOBS[Axle_Sensor-1])          /* Check current sensor status */
        {
            case IDLE: if(Timer1_Status == AVAIL)
            {
                TOBS[Axle_Sensor] = TIMER1;
                Timer1_Status = BUSY;
                Setup_Timer1();
            }
            else
            {
                if(Timer2_Status == AVAIL)
                {
                    TOBS[Axle_Sensor] = TIMER2;
                    Timer2_Status = BUSY;
                    Setup_Timer2();
                }
            }
            break;
            case TIMER1:TCR1 = 0x80;          /* Disable Timer 1 */
                Time1 = 0xFFFF-(((TCRH1<<8)+TCRM1)<<8)+TCRL1;
                /* Store timer 1 value */
                Timer1_Status = BUSY;
                TOBS[Axle_Sensor] = TIMER1;
                TOBS[Axle_Sensor-1] = IDLE;
                Setup_Timer1();              /* Enable Timer 1 */
                break;
            case TIMER2:TCR2 = 0x80;          /* Disable Timer 2 */
                if(Timer1_Status == AVAIL)
                {
                    /* Swap timer */
                    Time1 = 0xFFFF-(((TCRH2<<8)+TCRM2)<<8)+TCRL2;
                    /* Store timer 2 value */
                    Timer1_Status = BUSY;
                    Timer2_Status = AVAIL;
                    TOBS[Axle_Sensor] = TIMER1;
                    TOBS[Axle_Sensor-1] = IDLE;
                    Setup_Timer1();          /* Enable Timer 1 */
                }
                else
                {
                    Time2 = 0xFFFF-(((TCRH2<<8)+TCRM2)<<8)+TCRL2;
                    /* Store timer 2 value */
                    Timer2_Status = BUSY;
                    TOBS[Axle_Sensor] = TIMER2;
                    TOBS[Axle_Sensor-1] = IDLE;
                    Setup_Timer2();          /* Enable timer 2 */
                }
            }
        }
    }
}

```

```

        break;
    /* End switch */
}

/*****
Look back routine for sensor 6 to 10
*****/
Service_Sensor6_10(Axle_Sensor)
int Axle_Sensor;
{
    int i,j;

    i = 1;

    if(!EndRec)
    {
        if((Count[Axle_Sensor] > Axle_Dist_Cnt)&&(Count[Axle_Sensor] < Count[0]))
        {
            while((Count[Axle_Sensor-i] <= Count[Axle_Sensor])&&(i <= Axle_Sensor))
                ++i;
            if(i < Axle_Sensor)
            {
                switch(TOBS[Axle_Sensor - i])
                {
                    case TIMER1: TCR1 = 0x30; /* Disable Timer 1 */
                        /* These are the 4 numbers used to determine axle spacings.
                           Example: axle sensor 8 just get hit
                               sensor 5 is the closest sensor last get hit
                               time taken to travel from sensor 4 to sensor 5
                               time elapsed since sensor 5 get hit */
                        AXLE_SPACE[dist][0] = Axle_Sensor;
                        AXLE_SPACE[dist][1] = Axle_Sensor - i;
                        AXLE_SPACE[dist][2] = (Time1*0.000000125*32);
                        /* 3 Mhz clock 5 bit prescaler */
                        AXLE_SPACE[dist][3] = (0xFFFFF-((((TCRH1<<3)+TCRH1)<<3)+TCRL1))*0.000000125*32;
                        ++Axle_Dist_Cnt;
                        Timer1_Status = AVAIL;
                        Time1 = 0;
                        TOBS[Axle_Sensor-i] = IDLE;
                        ++dist;
                        break;
                    case TIMER2: TCR2 = 0x30; /* Disable Timer 2 */
                        AXLE_SPACE[dist][0] = Axle_Sensor;
                        AXLE_SPACE[dist][1] = Axle_Sensor - i;
                        AXLE_SPACE[dist][2] = (Time2*0.000000125*32);
                        AXLE_SPACE[dist][3] = (0xFFFFF-((((TCRH2<<3)+TCRH2)<<3)+TCRL2))*0.000000125*32;
                        ++Axle_Dist_Cnt;
                        Timer2_Status = AVAIL;
                        Time2 = 0;
                        TOBS[Axle_Sensor-i] = IDLE;
                        ++dist;
                        break;
                    default: puts("Can't find timer of lock back sensor");
                        Count[7] = 0;
                        Count[8] = 0;
                        Count[9] = 0;
                        Count[10] = 0;
                        Prev_Total_Ax_Count = 1;
                        break;
                }
            }
        }
    }
}

```

```

    /* End switch */

/* Write raw data to ram disk if so desire */
fprintf(fp1, "%d", Weight);
for(j = 0; j < 2; j++)
    fprintf(fp1, "%d", (int)AXLE_SPACE[dist-1][j]);
for(j = 2; j < 4; j++)
    fprintf(fp1, "%1.4f", AXLE_SPACE[dist-1][j]);
WEIGHT[dist-1] = WEIGHT_LEFT[dist-1] + WEIGHT_RIGHT[dist-1];
fprintf(fp1, "%d\n", WEIGHT[dist-1]);

/* End if */
/* End if */
/* End if !EndRec */
}
/*****
Read Weight(index, channel)
int index, channel;
{
    unsigned short *dataptr;
    unsigned short weight_value;
    int datcnt;

    dataptr = &weight_value;
    datcnt = 1;

    if(_rd_VDADi(pdadl, dataptr, datcnt, channel) == -1)
        _errmsg(errno, " can't read %s\n", dadlin);

    if(channel)
        /* If channel is 1 means weight pad
         at sensor 3 gets hit */
        WEIGHT_LEFT[index] = ( weight_value - 4096 ) * 16260 / 4096;
        /* 4096 is compensation for channel 1
         designation no compensation required
         for channel 0 */
        /* 16260 is scale correction factor
         interpolated from repeated measurement */
    else
        /* Read weight pad at sensor 1 */
        WEIGHT_RIGHT[index] = weight_value * 16420 / 4096;

/* Generate a pulse to reset peak hold circuitry */
if(channel)
{
    if (_ss_PBDIX_cntl(din_nun7, 0) == -1)
        exit(_errmsg(errno, " can't negate H2 on %s\n", din_nun7));
    fprintf (" vt %d\n", Count[3]);
    fprintf (" reset ");
    if (_ss_PBDIX_cntl(din_nun7, 1) == -1)
        exit(_errmsg(errno, " can't assert H2 on %s\n", din_nun7));
}
else
{
    if (_ss_PBDIX_cntl(din_nun8, 0) == -1)
        exit(_errmsg(errno, " can't negate H4 on %s\n", din_nun8));
    fprintf (" vt %d\n", Count[1]);
    fprintf (" reset ");
    if (_ss_PBDIX_cntl(din_nun8, 1) == -1)
        exit(_errmsg(errno, " can't assert H4 on %s\n", din_nun8));
}
}

```

THIS PAGE INTENTIONALLY BLANK

```

/*****
Compute.c
*****/

#include <stdio.h>
#include <time.h>
#include "trkrec.h"

extern FILE *fp3;
extern int Total_Ax_Count;

static float Pos[2][11] = { {0,2,4.02,6.02,8.02,10.04,12,22.07,32.07,42.04,52.11},
                             {2,2,2.02,2,2,2.02,1.96,10.07,10,9.97,10.07} };

time_t qtime;

/*****
      Get Axle Spacings Function
*****/
Get_Ax_Spacings()
(
    int index;
    int temp1,temp2;

/* initialize variables */

    axle_cnt = 1;
    axle_total = 0;
    data = 0;
    Ds = 0;
    as = 0;
    tr = 0.0;
    blv_count = 0;
    speed = 0.0;
    total_speed = 0.0;
    two_ft_sensor = MAXTHOFT;
    ten_ft_sensor = REFSSENSOR;
    sapltime = 0;
    axwt_index = 0; /* index for acquisition of individual */
                  /* axle weights, set to 0th element initially */
    weigh_pad = 1; /* weigh pad element in win_st axwt array */
    class = 0;
    truck_rec = 0;
    Invalid_flag = FALSE;

    if(Invalid_flag == TRUE){

        printf("*****\n");
        printf("*****\n");
        printf("***** INVALID READING *****\n");
        printf("*****\n");
        printf("*****\n");
    }
    else{

        /* Otherwise read AXLE data */

        truck_rec++;

```

```

for(axleindex = 0; axleindex < MAXAXLE; axleindex++){
    Recs[truck_rec].axwt[axleindex] = WEIGHT[axleindex];

for(axleindex = 0; axleindex < MAXAXLE; axleindex++){

if(AXLE_SPACE[axleindex][data] == 0){
    SpacingTotals(Recs, truck_rec); /* Axle spacings of all combinations */
    WeightTotals(TrkRec, Recs, truck_rec); /* Weights of all axle combinations */
    veh_class = Classification(Recs, truck_rec, axle_total); /* 1-15 */
    class = veh_class;
    avg_speed = (total_speed/(axle_total -1)); /* minus 1 -> axle total > axle pairs */
    speed = avg_speed*0.6818181818;

    time(&ctime);
    printf("*****\n");
    fprintf(fp3, "*****\n");
    printf("Site : Antelope      Date/Time: %s \n", ctime(&ctime));
    fprintf(fp3, "Site : Antelope      Date/Time: %s \n", ctime(&ctime));
    printf("Vehicle Number: %d      Class: %d      Average Speed: %4.2f mph\n",
        ++Id, class, speed);
    fprintf(fp3, "Vehicle Number: %d      Class: %d      Average Speed: %4.2f mph\n",
        Id, class, speed);
    printf("-----\n");
    fprintf(fp3, "-----\n");
    printf("Axle:      1      2      3      4      5      6      7      8      9\n");
    fprintf(fp3, "Axle:      1      2      3      4      5      6      7      8      9\n");
    printf("Weight: ");
    fprintf(fp3, "Weight: ");
    for(axleindex = 0; axleindex < axle_total; axleindex++) {
        printf("%8ld", Recs[truck_rec].axwt[axleindex]);
        fprintf(fp3, "%8ld", Recs[truck_rec].axwt[axleindex]);
    }
    printf("\n\nSpacing:      ");
    fprintf(fp3, "\n\nSpacing:      ");
    for(axleindex = 0; axleindex < (axle_total-1); axleindex++) {
        printf("%5.2f ", Recs[truck_rec].axsp_tot[axleindex]);
        fprintf(fp3, "%5.2f ", Recs[truck_rec].axsp_tot[axleindex]);
    }
    printf("\n-----Violation Summary-----\n");
    fprintf(fp3, "\n-----Violation Summary-----\n");
    WeightViolation(Recs, truck_rec);

/* Steering Axle Violation */

if(steer_wt != 0) {
    printf("Steering Axle >12500: %ld\n", steer_wt);
    fprintf(fp3, "Steering Axle >12500: %ld\n", steer_wt);
}

/* Single Axle Violations */

for(axleindex = 1; axleindex <= (MAXAXLE -1); axleindex++){
    if(SingleAxle[axleindex] != 0)
        print = 1;
}
if(print != 0) {
    printf("\nSingle Axle >20000\n");
    fprintf(fp3, "\nSingle Axle >20000\n");
    printf("(axle/weight):      ");
    fprintf(fp3, "(axle/weight):      ");
}

```

```

        print = 0;
        for(axleindex = 1; axleindex <= (MAXAXLE -1); axleindex++){
            if(SingleAxle[axleindex] != 0) {
                printf(" %d/%ld ",(axleindex +1),SingleAxle[axleindex]);
                fprintf(fp3," %d/%ld ",(axleindex +1),SingleAxle[axleindex]);
            }
        }
    }

/* Tandem Axle Violations */

    for(axleindex = 0; axleindex < TANDEMGROUPS; axleindex++){
        if(Tandem[axleindex] != 0)
            print = 1;
    }
    if(print != 0){
        printf("\n\nTandem Axle >34000\n");
        fprintf(fp3,"\n\nTandem Axle >34000\n");
        printf("(Tandem/weight):  ");
        fprintf(fp3,"(Tandem/weight):  ");
        print = 0;
        for(axleindex = 0; axleindex < TANDEMGROUPS; axleindex++){
            if(Tandem[axleindex] != 0) {
                printf("%d-%d/%ld ",(axleindex +1),(axleindex +2),Tandem[axleindex]);
                fprintf(fp3,"%d-%d/%ld ",(axleindex +1),(axleindex +2),Tandem[axleindex]);
            }
        }
    }

/* Gross Weight Violations */

    if(gross_wt != 0){
        printf("\n\nGross Weight >80000:  %ld",gross_wt);
        fprintf(fp3,"\n\nGross Weight >80000:  %ld",gross_wt);
    }

/* Bridge Lav Violations */

    for(axleindex = 0; axleindex < MAXGROUPS; axleindex++){
        if(BridgeViolate[axleindex] != 0){
            blv_count++;
        }
    }
    printf("\n\nBridge Lav Violations: %d\n",blv_count);
    fprintf(fp3,"\n\nBridge Lav Violations: %d\n",blv_count);
    printf("*****\n");
} /* End if AXLE_SPACE[axleindex][data] = 0 */
else{

/* Otherwise collect more axle data */

    data = 0;
    SS = AXLE_SPACE[axleindex][data];
    data++;
    So = AXLE_SPACE[axleindex][data];
    data++;
    tr = AXLE_SPACE[axleindex][data];
    data++;
    to = AXLE_SPACE[axleindex][data];

```



```

/* Calculate axle spacing */

if(tr == 0.0) { puts("OOP"); tr = 1.0; }
V = (Pos[1][So]*100)/(tr*100); /* speed (ft/sec) */
Vmph = V*0.6818181818;
printf("Speed = %4.2f mph\n",Vmph);
total_speed = total_speed + V;
Ds = V*to; /*distance traveled since ref. sensor hit */
as = Pos[0][SS] - Pos[0][So] - Ds; /*ref sensor - offset sensor - Ds */

Recs[truck_rec].axsp_tot[(axle_cnt-1)] = as; /*store as in truck_rec array*/
axle_total = (axle_cnt + 1);
axle_cnt++;
} /* End else */
} /* End for loop */
} /* End else */
} /* End Get_Ax_Spacings Function */

/*****
SpacingTotals Function
*****/
/* This function calculates the spacings of all */
/* axle combinations. */
/*****/

float *SpacingTotals(RecsPtr, truck_rec)
struct win_st *RecsPtr;

{
int space_tot_initial; /* Sets starting position in axsp array */
int space_tot_final; /* Sets ending position in axsp array */
int axsp_tot_index; /* Index for axsp_tot array (holds all */
/* axle grouping combinations) */
int modifier; /* adjusts number of loops per axle grouping */
int offset; /* sets lower limit on number of loops per axle grouping */
int initial_offset;
int addoffset;
int aoffset;

axsp_tot_index = 7; /* index values 0-6 filled by basic axle- */
/* spacing claulation (above). */
addoffset = 0;
modifier = 7; /* adjusts number of loops per axle grouping */

space_tot_initial = 0; /* counters for each set of axle groupings */
space_tot_final = 6; /* starting with axle sets of 3 (1-3 to 7-9) */
/* -> 7 groups or 0-6 times thru the loop */

for (offset = modifier; offset >= 1; offset--){
addoffset++;
aoffset = addoffset;
for (initial_offset = space_tot_initial; initial_offset <=
space_tot_final; initial_offset++){
++axsp_tot_index;
RecsPtr[truck_rec].axsp_tot[axsp_tot_index] =
RecsPtr[truck_rec].axsp_tot[space_tot_initial] +
RecsPtr[truck_rec].axsp_tot[aoffset];
if(RecsPtr[truck_rec].axsp_tot[aoffset] == 0){
RecsPtr[truck_rec].axsp_tot[axsp_tot_index] = 0;
}
}
}

```

```

        aoffset++;
        space_tot_initial++;
    }
    space_tot_initial = (space_tot_final + 2);
    space_tot_final = (space_tot_final + offset);
}
return(RecsPtr);
}
/*****
                        WeightTotals Function
*****/
/*          This function calculates the weight of all          */
/*          axle combinations.                                  */
/*****

long *WeightTotals(TrkRec,PtrRecs,truck_rec)
struct win_st *PtrRecs;
long TrkRec[SMPLTIME][SENSOR];
{

    int group_index;      /* index used for each axle group */
    int x_axle_index;     /* 2 axle group index */
    int y_axle_index;     /* index for all other axle groupings */
    int offset_a;         /* determines initial position y_axle addition */
    int offset_b;         /* determines initial y_count value for each new loop */
    int axwt_tot_count;   /* Count for axwt tot array */
    int y_axle_ref_index; /* place holder for y_axle count */
    int maxaxles;         /* upper limit for 2 axle groups (0-8 => 9 axles) */
    int minloop;          /* lower limit for 2 axle groups */

    axwt_tot_count = 0;   /* array counter for 0-35 elements */
    maxaxles = 8;         /* 0-8 => 9 axles */
    offset_a = -11;       /* offsets are magic numbers needed to make */
    offset_b = -5;        /* this routine work - see documentation */
    minloop = 2;          /* min number of times thru loop */

    /* Individuale axle weights pulled in from TrkRec directly */
    /* scan through weigh pad array for weights */

    for (wp_smpltime = 0; wp_smpltime < SMPLTIME; wp_smpltime++){
        if(TrkRec[wp_smpltime][weigh_pad] != 0){
            Recs[truck_rec].axwt[axwt_index] = TrkRec[wp_smpltime][weigh_pad];
            Axwt_time[axwt_index] = wp_smpltime;
            axwt_index++;
        }
    }

    /* loop for all 2 axle combinations */

    for (x_axle_index = 0; x_axle_index < maxaxles; x_axle_index++){
        Recs[truck_rec].axwt_tot[axwt_tot_count] =
            (Recs[truck_rec].axwt[x_axle_index]) + (Recs[truck_rec].axwt[(x_axle_index + 1)]);
        axwt_tot_count++;
    }

    /* Loop for all other axle combinations. First "for" loop */
    /* sets number of repetitions for each axle group */

```

```

for (group_index = maxaxles; group_index >= minloop; group_index--){
    offset_a = offset_a + group_index;
    y_axle_index = group_index + offset_b;
    offset_b = offset_b + minloop;
    y_axle_ref_index = y_axle_index;

    /* Second 'for' loop sets the array element numbers to be */
    /* added in each axle group calculation (i.e. axles 1-7 add */
    /* the sum of 1-6 and axle 7 */

    for(y_axle_index = y_axle_ref_index; y_axle_index <= MAXAXLE; ++y_axle_index){
        Recs[truck_rec].axwt_tot[axwt_tot_count] =
            Recs[truck_rec].axwt_tot[(y_axle_index + offset_a)] +
            Recs[truck_rec].axwt[(y_axle_index - 1)];
        axwt_tot_count++;
    }
}
return (PtrRecs);

```

```

}
/*****
Weight Violation Function
*****/
/*      This function calculates the weight violations for      */
/*      all axle combinations (using bridge law and weight tables).  */
/*****

```

```

long *WeightViolation(VioPtr, vio_truck_rec)
struct vio_st *VioPtr;

```

```

{

```

```

    int axwt_tot_count;
    int tandem_index;
    int groupindex;
    int spaceindex;

```

```

static long BridgeLaw[76][10] = {
    {0,0,34000,34000,34000,34000,34000,34000,34000,34000},
    {1,0,34000,34000,34000,34000,34000,34000,34000,34000},
    {2,0,34000,34000,34000,34000,34000,34000,34000,34000},
    {3,0,34000,34000,34000,34000,34000,34000,34000,34000},
    {4,0,34000,34000,34000,34000,34000,34000,34000,34000},
    {5,0,34000,34000,34000,34000,34000,34000,34000,34000},
    {6,0,34000,34000,34000,34000,34000,34000,34000,34000},
    {7,0,34000,34000,34000,34000,34000,34000,34000,34000},
    {8,0,34000,34000,34000,34000,34000,34000,34000,34000},
    {9,0,39000,42500,42500,42500,42500,42500,42500,42500},
    {10,0,40000,43500,43500,43500,43500,43500,43500,43500},
    {11,0,40000,44000,44000,44000,44000,44000,44000,44000},
    {12,0,40000,45000,50000,50000,50000,50000,50000,50000},
    {13,0,40000,45500,50500,50500,50500,50500,50500,50500},
    {14,0,40000,46500,51500,51500,51500,51000,51000,51000},
    {15,0,40000,47000,52000,52000,52000,52000,52000,52000},
    {16,0,40000,48000,52500,52500,52500,52500,52500,52500},
    {17,0,40000,48500,53500,53500,53500,53500,53500,53500},
    {18,0,40000,49500,54000,54000,54000,54500,54500,54500},
    {19,0,40000,50000,54500,54500,54500,54500,54500,54500},
    {20,0,40000,51000,55500,55500,55500,55500,55500,55500},

```

```

{21,0,40000,51500,56000,56000,56000,56000,56000,56000},
{22,0,40000,52500,56500,56500,56500,56500,56500,56500},
{23,0,40000,53000,57500,57500,57500,57500,57500,57500},
{24,0,40000,54000,58000,58000,58000,74000,74000,74000},
{25,0,40000,54500,58500,58500,58500,74500,80000,80000},
{26,0,40000,55500,59500,59500,59500,75000,80000,80000},
{27,0,40000,56000,60000,60000,60000,76000,80000,80000},
{28,0,40000,57000,60500,60500,60500,76500,80000,80000},
{29,0,40000,57500,61500,61500,61500,77000,80000,80000},
{30,0,40000,58500,62000,62000,62000,77500,80000,80000},
{31,0,40000,59000,62500,62500,62500,78000,80000,80000},
{32,0,40000,60000,63500,63500,63500,78500,80000,80000},
{33,0,40000,60000,64000,64000,64000,79500,80000,80000},
{34,0,40000,60000,64500,64500,64500,80000,80000,80000},
{35,0,40000,60000,65500,65500,65500,80000,80000,80000},
{36,0,40000,60000,68000,66000,66000,80000,80000,80000},
{37,0,40000,60000,68000,66500,66500,80000,80000,80000},
{38,0,40000,60000,68000,67500,67500,80000,80000,80000},
{39,0,40000,60000,68000,68000,68000,80000,80000,80000},
{40,0,40000,60000,68500,70000,70000,80000,80000,80000},
{41,0,40000,60000,69500,72000,72000,80000,80000,80000},
{42,0,40000,60000,70000,73280,73280,80000,80000,80000},
{43,0,40000,60000,70500,73280,73280,80000,80000,80000},
{44,0,40000,60000,71500,73280,73280,80000,80000,80000},
{45,0,40000,60000,72000,76000,80000,80000,80000,80000},
{46,0,40000,60000,72500,76500,80000,80000,80000,80000},
{47,0,40000,60000,73500,77500,80000,80000,80000,80000},
{48,0,40000,60000,74000,78000,80000,80000,80000,80000},
{49,0,40000,60000,74500,78500,80000,80000,80000,80000},
{50,0,40000,60000,75500,79000,80000,80000,80000,80000},
{51,0,40000,60000,76000,80000,80000,80000,80000,80000},
{52,0,40000,60000,76500,80000,80000,80000,80000,80000},
{53,0,40000,60000,77500,80000,80000,80000,80000,80000},
{54,0,40000,60000,78000,80000,80000,80000,80000,80000},
{55,0,40000,60000,78500,80000,80000,80000,80000,80000},
{56,0,40000,60000,79500,80000,80000,80000,80000,80000},
{57,0,40000,60000,80000,80000,80000,80000,80000,80000},
{58,0,40000,60000,80000,80000,80000,80000,80000,80000},
{59,0,40000,60000,80000,80000,80000,80000,80000,80000},
{60,0,40000,60000,80000,80000,80000,80000,80000,80000},
{61,0,40000,60000,80000,80000,80000,80000,80000,80000},
{62,0,40000,60000,80000,80000,80000,80000,80000,80000},
{63,0,40000,60000,80000,80000,80000,80000,80000,80000},
{64,0,40000,60000,80000,80000,80000,80000,80000,80000},
{65,0,40000,60000,80000,80000,80000,80000,80000,80000},
{66,0,40000,60000,80000,80000,80000,80000,80000,80000},
{67,0,40000,60000,80000,80000,80000,80000,80000,80000},
{68,0,40000,60000,80000,80000,80000,80000,80000,80000},
{69,0,40000,60000,80000,80000,80000,80000,80000,80000},
{70,0,40000,60000,80000,80000,80000,80000,80000,80000},
{71,0,40000,60000,80000,80000,80000,80000,80000,80000},
{72,0,40000,60000,80000,80000,80000,80000,80000,80000},
{73,0,40000,60000,80000,80000,80000,80000,80000,80000},
{74,0,40000,60000,80000,80000,80000,80000,80000,80000},
{75,0,40000,60000,80000,80000,80000,80000,80000,80000},
];

```

```

gross_wt = 0;
steer_wt = 0; /* holds overweight value for steering axle, */

```

```

        /* remains zero if steering axle not overweight */
spaceindex = 0;
groupindex = 0;

/* Check steering axle for violation */
if(Recs[truck_rec].axwt[0] > STEERINGMAX){
    steer_vt = Recs[truck_rec].axwt[0];
}

/* Sets Tandem[] array to all zero values */
for(tandem_index = 0; tandem_index < TANDENGROUPS; tandem_index++){
    Tandem[tandem_index] = 0;
}

/* Check all tandem combinations for violations */
for(tandem_index = 0; tandem_index < TANDENGROUPS; tandem_index++){
    if(Recs[truck_rec].axsp_tot[tandem_index] < TANDEM_DEF){
        if(Recs[truck_rec].axsp_tot[tandem_index] != 0){
            if(Recs[truck_rec].axwt_tot[tandem_index] > TANDEMMAX){
                Tandem[tandem_index] =
                    Recs[truck_rec].axwt_tot[tandem_index];
            }
        }
    }
}

/* Check all other single axles for violations */
/*      printf("\nSingle_Weight"); */

for(groupindex = 0; groupindex <= (MAXAXLE - 1); groupindex++){
    SingleAxle[groupindex] = 0;

    if(Recs[truck_rec].axwt[groupindex] > SINGLEMAX){
        SingleAxle[groupindex] =
            Recs[truck_rec].axwt[groupindex];
    }
}

/* Check gross vehicle weight for violation */

if(Recs[truck_rec].axwt_tot[MAXGROUPS-1] > GROSSMAX){
    gross_vt = Recs[truck_rec].axwt_tot[MAXGROUPS-1];
}

/* Bridge Law Violation Check */
/* Set BridgeViolate array to 0 */

for(groupindex = 0; groupindex < MAXGROUPS; groupindex++){
    BridgeViolate[groupindex] = 0;
}

for(groupindex = 0; groupindex < MAXGROUPS; groupindex++){
    for(spaceindex = 8; spaceindex <= 58; spaceindex++){

        /* 2 axle grouping check */
        /* First 2 axle groups with spacings < 8 ft */

```

```

if(groupindex >= 0 && groupindex <= 7){
  if(Recs[truck_rec].axsp_tot[groupindex] < (7.5)){
    if(Recs[truck_rec].axwt_tot[groupindex] > BridgeLaw[spaceindex][2]){
      BridgeViolate[groupindex] = Recs[truck_rec].axwt_tot[groupindex];
    }
  }
  if(Recs[truck_rec].axsp_tot[groupindex] >= (spaceindex - 0.5) &&
    Recs[truck_rec].axsp_tot[groupindex] <= (spaceindex + 0.5)){
    if(Recs[truck_rec].axwt_tot[(groupindex)] >=
      BridgeLaw[spaceindex][2]){
      BridgeViolate[groupindex] =
        Recs[truck_rec].axwt_tot[groupindex];
    }
  }
}

/* 3 axle groupings */

if(groupindex >= 8 && groupindex <= 14){
  if(Recs[truck_rec].axsp_tot[groupindex] >= (spaceindex - 0.5) &&
    Recs[truck_rec].axsp_tot[groupindex] <=
      (spaceindex + 0.5)){
    if(Recs[truck_rec].axwt_tot[(groupindex)] >=
      BridgeLaw[spaceindex][3]){
      BridgeViolate[groupindex] =
        Recs[truck_rec].axwt_tot[groupindex];
    }
  }
}

/* 4 axle groupings */

if(groupindex >= 15 && groupindex <= 20){
  if(Recs[truck_rec].axsp_tot[groupindex] >= (spaceindex - 0.5) &&
    Recs[truck_rec].axsp_tot[groupindex] <=
      (spaceindex + 0.5)){
    if(Recs[truck_rec].axwt_tot[(groupindex)] >=
      BridgeLaw[spaceindex][4]){
      BridgeViolate[groupindex] =
        Recs[truck_rec].axwt_tot[groupindex];
    }
  }
}

/* 5 axle groupings */

if(groupindex >= 21 && groupindex <= 25){
  if(Recs[truck_rec].axsp_tot[groupindex] >= (spaceindex - 0.5) &&
    Recs[truck_rec].axsp_tot[groupindex] <=
      (spaceindex + 0.5)){
    if(Recs[truck_rec].axwt_tot[(groupindex)] >=
      BridgeLaw[spaceindex][5]){
      BridgeViolate[groupindex] =
        Recs[truck_rec].axwt_tot[groupindex];
    }
  }
}

/* 6 axle groupings */

```

```

if(groupindex >= 26 && groupindex <= 29){
  if(Recs[truck_rec].axsp_tot[groupindex] >= (spaceindex - 0.5) &&
    Recs[truck_rec].axsp_tot[groupindex] <=
      (spaceindex + 0.5)){
    if(Recs[truck_rec].axwt_tot[(groupindex)] >=
      BridgeLav[spaceindex][6]){
      BridgeViolate[groupindex] =
        Recs[truck_rec].axwt_tot[groupindex];
    }
  }
}

/* 7 axle groupings */

if(groupindex >= 30 && groupindex <= 32){
  if(Recs[truck_rec].axsp_tot[groupindex] >= (spaceindex - 0.5) &&
    Recs[truck_rec].axsp_tot[groupindex] <=
      (spaceindex + 0.5)){
    if(Recs[truck_rec].axwt_tot[(groupindex)] >=
      BridgeLav[spaceindex][7]){
      BridgeViolate[groupindex] =
        Recs[truck_rec].axwt_tot[groupindex];
    }
  }
}

/* 8 axle groupings */

if(groupindex >= 33 && groupindex <= 34){
  if(Recs[truck_rec].axsp_tot[groupindex] >= (spaceindex - 0.5) &&
    Recs[truck_rec].axsp_tot[groupindex] <=
      (spaceindex + 0.5)){
    if(Recs[truck_rec].axwt_tot[(groupindex)] >=
      BridgeLav[spaceindex][8]){
      BridgeViolate[groupindex] =
        Recs[truck_rec].axwt_tot[groupindex];
    }
  }
}

/* 9 axle groupings */

if(groupindex == 35){
  if(Recs[truck_rec].axsp_tot[groupindex] >= (spaceindex - 0.5) &&
    Recs[truck_rec].axsp_tot[groupindex] <=
      (spaceindex + 0.5)){
    if(Recs[truck_rec].axwt_tot[(groupindex)] >=
      BridgeLav[spaceindex][9]){
      BridgeViolate[groupindex] =
        Recs[truck_rec].axwt_tot[groupindex];
    }
  }
} /* end axle grouping loop check */
} /* end axle group spacing loop check */
} /* end bridge lav scan loop */
}

/*****
Classification Function
*****/

```

```

/*****
/*      This function determines the class of each vehicle      */
/*****

```

```

int Classification(ClassPtr,class_truck_rec,class_axle_total)
struct win_st *ClassPtr;

```

```

{
int classindex;

```

```

switch(class_axle_total){

```

```

/***** Two axles classes *****/

```

```

case 2: if((Recs[truck_rec].axsp_tot[0] >= 1.0) &&
(Recs[truck_rec].axsp_tot[0] <= 6.0) &&
(Recs[truck_rec].axwt_tot[(MAXGROUPS)] >= 100) &&
(Recs[truck_rec].axwt_tot[(MAXGROUPS)] <= 3000))
class = 1;

```

```

else

```

```

if((Recs[truck_rec].axsp_tot[0] >= 6.1) &&
(Recs[truck_rec].axsp_tot[0] <= 9.9) &&
(Recs[truck_rec].axwt_tot[(MAXGROUPS)] >= 1000) &&
(Recs[truck_rec].axwt_tot[(MAXGROUPS)] <= 7990))
class = 2;

```

```

else

```

```

if((Recs[truck_rec].axsp_tot[0] >= 10.0) &&
(Recs[truck_rec].axsp_tot[0] <= 14.5) &&
(Recs[truck_rec].axwt_tot[(MAXGROUPS)] >= 1000) &&
(Recs[truck_rec].axwt_tot[(MAXGROUPS)] <= 7990))
class = 3; -

```

```

else

```

```

if((Recs[truck_rec].axsp_tot[0] >= 23.1) &&
(Recs[truck_rec].axsp_tot[0] <= 40.0))
class = 4;

```

```

else

```

```

if((Recs[truck_rec].axsp_tot[0] >= 8.8) &&
(Recs[truck_rec].axsp_tot[0] <= 23.0) &&
(Recs[truck_rec].axwt_tot[(MAXGROUPS)] >= 8000))
class = 5;

```

```

else

```

```

class = 15;

```

```

break;

```

```

/***** Three axle classes *****/

```

```

case 3: if((Recs[truck_rec].axsp_tot[0] >= 6.1) &&
(Recs[truck_rec].axsp_tot[0] <= 9.9) &&
(Recs[truck_rec].axsp_tot[1] >= 6.0) &&
(Recs[truck_rec].axsp_tot[1] <= 25.0) &&
(Recs[truck_rec].axwt_tot[(MAXGROUPS)] >= 1000) &&
(Recs[truck_rec].axwt_tot[(MAXGROUPS)] <= 11990))
class = 2;

```

```

else

```

```

if((Recs[truck_rec].axsp_tot[0] >= 10.0) &&
(Recs[truck_rec].axsp_tot[0] <= 14.5) &&
(Recs[truck_rec].axsp_tot[1] >= 6.0) &&
(Recs[truck_rec].axsp_tot[1] <= 25.0) &&
(Recs[truck_rec].axwt_tot[(MAXGROUPS)] >= 1000) &&
(Recs[truck_rec].axwt_tot[(MAXGROUPS)] <= 11990))
class = 3;

```

```

else

```



```

if((Recs[truck_rec].axsp_tot[0] >= 23.1) &&
  (Recs[truck_rec].axsp_tot[0] <= 40.0) &&
  (Recs[truck_rec].axsp_tot[1] >= 3.5) &&
  (Recs[truck_rec].axsp_tot[1] <= 6.0))
  class = 4;
else
  if((Recs[truck_rec].axsp_tot[0] >= 6.1) &&
    (Recs[truck_rec].axsp_tot[0] <= 23.0) &&
    (Recs[truck_rec].axsp_tot[1] >= 3.5) &&
    (Recs[truck_rec].axsp_tot[1] <= 6.0))
    class = 6;
  else
    if((Recs[truck_rec].axsp_tot[0] >= 6.1) &&
      (Recs[truck_rec].axsp_tot[0] <= 23.0) &&
      (Recs[truck_rec].axsp_tot[1] >= 11.0) &&
      (Recs[truck_rec].axsp_tot[1] <= 40.0) &&
      (Recs[truck_rec].axwt_tot[(MAXGROUPS)] >= 12000))
      class = 8;
    else
      class = 15;
  break;
/***** Four axle classes *****/
case 4: if((Recs[truck_rec].axsp_tot[0] >= 6.1) &&
  (Recs[truck_rec].axsp_tot[0] <= 9.9) &&
  (Recs[truck_rec].axsp_tot[1] >= 6.0) &&
  (Recs[truck_rec].axsp_tot[1] <= 25.0) &&
  (Recs[truck_rec].axsp_tot[2] >= 1.0) &&
  (Recs[truck_rec].axsp_tot[2] <= 3.4) &&
  (Recs[truck_rec].axwt_tot[(MAXGROUPS)] >= 1000) &&
  (Recs[truck_rec].axwt_tot[(MAXGROUPS)] <= 11990))
  class = 2;
else
  if((Recs[truck_rec].axsp_tot[0] >= 10.0) &&
    (Recs[truck_rec].axsp_tot[0] <= 14.5) &&
    (Recs[truck_rec].axsp_tot[1] >= 6.0) &&
    (Recs[truck_rec].axsp_tot[1] <= 25.0) &&
    (Recs[truck_rec].axsp_tot[2] >= 1.0) &&
    (Recs[truck_rec].axsp_tot[2] <= 3.4) &&
    (Recs[truck_rec].axwt_tot[(MAXGROUPS)] >= 1000) &&
    (Recs[truck_rec].axwt_tot[(MAXGROUPS)] <= 11990))
    class = 3;
  else
    if((Recs[truck_rec].axsp_tot[0] >= 6.1) &&
      (Recs[truck_rec].axsp_tot[0] <= 23.0) &&
      (Recs[truck_rec].axsp_tot[1] >= 3.5) &&
      (Recs[truck_rec].axsp_tot[1] <= 6.0) &&
      (Recs[truck_rec].axsp_tot[2] >= 3.5) &&
      (Recs[truck_rec].axsp_tot[2] <= 6.0))
      class = 7;
    else
      if(((Recs[truck_rec].axsp_tot[0] >= 6.1) &&
        (Recs[truck_rec].axsp_tot[0] <= 23.0) &&
        (Recs[truck_rec].axsp_tot[1] >= 3.5) &&
        (Recs[truck_rec].axsp_tot[1] <= 6.0) &&
        (Recs[truck_rec].axsp_tot[2] >= 6.1) &&
        (Recs[truck_rec].axsp_tot[2] <= 40.0) &&
        (Recs[truck_rec].axwt_tot[(MAXGROUPS)] >= 12000)))||
        ((Recs[truck_rec].axsp_tot[0] >= 6.1) &&
        (Recs[truck_rec].axsp_tot[0] <= 23.0) &&
        (Recs[truck_rec].axsp_tot[1] >= 11.0) &&

```

```

        (Recs[truck_rec].axsp_tot[1] <= 40.0) &&
        (Recs[truck_rec].axsp_tot[2] >= 3.5) &&
        (Recs[truck_rec].axsp_tot[2] <= 10.9) &&
        (Recs[truck_rec].axwt_tot[(MAXGROUPS)] >= 12000)))
        class = 8;
    else
        class = 15;

    break;
/***** Five axle classes *****/
case 5: if((Recs[truck_rec].axsp_tot[0] >= 10.0) &&
        (Recs[truck_rec].axsp_tot[0] <= 14.5) &&
        (Recs[truck_rec].axsp_tot[1] >= 6.0) &&
        (Recs[truck_rec].axsp_tot[1] <= 25.0) &&
        (Recs[truck_rec].axsp_tot[2] >= 1.0) &&
        (Recs[truck_rec].axsp_tot[2] <= 3.4) &&
        (Recs[truck_rec].axsp_tot[3] >= 1.0) &&
        (Recs[truck_rec].axsp_tot[3] <= 3.4) &&
        (Recs[truck_rec].axwt_tot[(MAXGROUPS)] >= 1000) &&
        (Recs[truck_rec].axwt_tot[(MAXGROUPS)] <= 11990))
        class = 3;
    else
        if((Recs[truck_rec].axsp_tot[0] >= 6.1) &&
            (Recs[truck_rec].axsp_tot[0] <= 26.0) &&
            (Recs[truck_rec].axsp_tot[1] >= 3.5) &&
            (Recs[truck_rec].axsp_tot[1] <= 6.0) &&
            (Recs[truck_rec].axsp_tot[2] >= 6.1) &&
            (Recs[truck_rec].axsp_tot[2] <= 46.0) &&
            (Recs[truck_rec].axsp_tot[3] >= 3.5) &&
            (Recs[truck_rec].axsp_tot[3] <= 10.9) &&
            (Recs[truck_rec].axwt_tot[(MAXGROUPS)] >= 12000))
            class = 9;
        else
            if((Recs[truck_rec].axsp_tot[0] >= 6.1) &&
                (Recs[truck_rec].axsp_tot[0] <= 26.0) &&
                (Recs[truck_rec].axsp_tot[1] >= 11.1) &&
                (Recs[truck_rec].axsp_tot[1] <= 26.0) &&
                (Recs[truck_rec].axsp_tot[2] >= 6.1) &&
                (Recs[truck_rec].axsp_tot[2] <= 20.0) &&
                (Recs[truck_rec].axsp_tot[3] >= 11.1) &&
                (Recs[truck_rec].axsp_tot[3] <= 26.0) &&
                (Recs[truck_rec].axwt_tot[(MAXGROUPS)] >= 12000))
                    class = 11;
            else
                if((Recs[truck_rec].axsp_tot[0] >= 6.1) &&
                    (Recs[truck_rec].axsp_tot[0] <= 23.0) &&
                    (Recs[truck_rec].axsp_tot[1] >= 3.5) &&
                    (Recs[truck_rec].axsp_tot[1] <= 6.0) &&
                    (Recs[truck_rec].axsp_tot[2] >= 6.1) &&
                    (Recs[truck_rec].axsp_tot[2] <= 23.0) &&
                    (Recs[truck_rec].axsp_tot[3] >= 11.1) &&
                    (Recs[truck_rec].axsp_tot[3] <= 27.0) &&
                    (Recs[truck_rec].axwt_tot[(MAXGROUPS)] >= 12000))
                        class = 14;
                else
                    class = 15;

            break;
/***** Six axle classes *****/
case 6: if((Recs[truck_rec].axsp_tot[0] >= 6.1) &&
        (Recs[truck_rec].axsp_tot[0] <= 26.0) &&

```

```

(Recs[truck_rec].axsp_tot[1] >= 3.5) &&
(Recs[truck_rec].axsp_tot[1] <= 6.0) &&
(Recs[truck_rec].axsp_tot[2] >= 6.1) &&
(Recs[truck_rec].axsp_tot[2] <= 46.0) &&
(Recs[truck_rec].axsp_tot[3] >= 1.0) &&
(Recs[truck_rec].axsp_tot[3] <= 11.0) &&
(Recs[truck_rec].axsp_tot[4] >= 1.0) &&
(Recs[truck_rec].axsp_tot[4] <= 11.0) &&
(Recs[truck_rec].axwt_tot[(MAXGROUPS)] >= 12000))
    class = 10;
else
if((Recs[truck_rec].axsp_tot[0] >= 6.1) &&
(Recs[truck_rec].axsp_tot[0] <= 26.0) &&
(Recs[truck_rec].axsp_tot[1] >= 3.5) &&
(Recs[truck_rec].axsp_tot[1] <= 6.0) &&
(Recs[truck_rec].axsp_tot[2] >= 11.1) &&
(Recs[truck_rec].axsp_tot[2] <= 26.0) &&
(Recs[truck_rec].axsp_tot[3] >= 6.1) &&
(Recs[truck_rec].axsp_tot[3] <= 20.0) &&
(Recs[truck_rec].axsp_tot[4] >= 11.1) &&
(Recs[truck_rec].axsp_tot[4] <= 26.0) &&
(Recs[truck_rec].axwt_tot[(MAXGROUPS)] >= 12000))
    class = 12;
else
    class = 15;
break;
/***** Seven Axle Classes *****/
case 7: if((Recs[truck_rec].axsp_tot[0] >= 1.0) &&
(Recs[truck_rec].axsp_tot[0] <= 45.0) &&
(Recs[truck_rec].axsp_tot[1] >= 1.0) &&
(Recs[truck_rec].axsp_tot[1] <= 45.0) &&
(Recs[truck_rec].axsp_tot[2] >= 1.0) &&
(Recs[truck_rec].axsp_tot[2] <= 45.0) &&
(Recs[truck_rec].axsp_tot[3] >= 1.0) &&
(Recs[truck_rec].axsp_tot[3] <= 45.0) &&
(Recs[truck_rec].axsp_tot[4] >= 1.0) &&
(Recs[truck_rec].axsp_tot[4] <= 45.0) &&
(Recs[truck_rec].axsp_tot[5] <= 1.0) &&
(Recs[truck_rec].axsp_tot[5] >= 45.0) &&
(Recs[truck_rec].axwt_tot[(MAXGROUPS)] >= 12000))
    class = 13;
else
    class = 15;
break;
/***** Right axle class *****/
case 8: if((Recs[truck_rec].axsp_tot[0] >= 1.0) &&
(Recs[truck_rec].axsp_tot[0] <= 45.0) &&
(Recs[truck_rec].axsp_tot[1] >= 1.0) &&
(Recs[truck_rec].axsp_tot[1] <= 45.0) &&
(Recs[truck_rec].axsp_tot[2] >= 1.0) &&
(Recs[truck_rec].axsp_tot[2] <= 45.0) &&
(Recs[truck_rec].axsp_tot[3] >= 1.0) &&
(Recs[truck_rec].axsp_tot[3] <= 45.0) &&
(Recs[truck_rec].axsp_tot[4] >= 1.0) &&
(Recs[truck_rec].axsp_tot[4] <= 45.0) &&
(Recs[truck_rec].axsp_tot[5] <= 1.0) &&
(Recs[truck_rec].axsp_tot[5] >= 45.0) &&
(Recs[truck_rec].axsp_tot[6] <= 1.0) &&
(Recs[truck_rec].axsp_tot[6] <= 45.0) &&
(Recs[truck_rec].axwt_tot[(MAXGROUPS)] >= 12000))

```

```

        class = 13;
    else
        class = 15;
        break;
/***** Nine axle class *****/
case 9: if((Recs[truck_rec].axsp_tot[0] >= 1.0) &&
    (Recs[truck_rec].axsp_tot[0] <= 45.0) &&
    (Recs[truck_rec].axsp_tot[1] >= 1.0) &&
    (Recs[truck_rec].axsp_tot[1] <= 45.0) &&
    (Recs[truck_rec].axsp_tot[2] >= 1.0) &&
    (Recs[truck_rec].axsp_tot[2] <= 45.0) &&
    (Recs[truck_rec].axsp_tot[3] >= 1.0) &&
    (Recs[truck_rec].axsp_tot[3] <= 45.0) &&
    (Recs[truck_rec].axsp_tot[4] >= 1.0) &&
    (Recs[truck_rec].axsp_tot[4] <= 45.0) &&
    (Recs[truck_rec].axsp_tot[5] <= 1.0) &&
    (Recs[truck_rec].axsp_tot[5] >= 45.0) &&
    (Recs[truck_rec].axsp_tot[6] <= 1.0) &&
    (Recs[truck_rec].axsp_tot[6] >= 45.0) &&
    (Recs[truck_rec].axsp_tot[7] <= 1.0) &&
    (Recs[truck_rec].axsp_tot[7] <= 45.0) &&
    (Recs[truck_rec].axwt_tot[(MAXGROUPS)] >= 12000))
        class = 13;
    else
        class = 15;
        break;
/***** Total Axle > 9 *****/
default: class = 15;
        break;
}
return(class);
}

```

THIS PAGE INTENTIONALLY BLANK

```

/*****
trkrec.h
*****/

```

This file defines all the variables, macros, structures, arrays, etc. needed for WIN main() and all the functions call from within main().

```

*****/

#define SENSOR 14
#define SMPLTIME 240
#define NUMTOCLEAR 25
#define MAXSENSOR 13
#define REFSSENSOR 8
#define MAXTWOFT 7
#define MINSSENSOR 4
#define SENSORPOS 14
#define TWOFTSPACING 2
#define SMPLCALFCTR 0.02
#define MINAXLE 1
#define MAXGROUPS 36 /*max axle grouping combinations for weight and spacing */
#define MAXAXLE 9
#define previous_sensor_cnt (axle cnt + 1)
#define MAXRECS 5 /*sizeof Recs/sizeof(struct win_st)*/
#define STEERINGMAX 12500
#define TANDENMAX 34000
#define SINGLEMAX 20000
#define GROSSMAX 80000
#define TANDENGROUPS 7 /* Number of tandem sets in Tandem array 0-6 => 7 */
#define TANDEM_DEF 8.4 /* Defines spacing for tandem set */
#define TRUE 0
#define FALSE 1

int SensorLocation[SENSORPOS];
int Axwt_time[MAXAXLE];
/* int Pos[2][10]; sensor position array */
long Tandem[TANDENGROUPS];
long SingleAxle[MAXGROUPS];
long TrkRec[SMPLTIME][SENSOR];
long BridgeViolate[MAXGROUPS];

float AXLE_SPACE[MAXAXLE][4];
long WEIGHT[MAXAXLE];
int WEIGHT_LEFT[MAXAXLE];
int WEIGHT_RIGHT[MAXAXLE];
int print;
float avg_speed;
float total_speed;

int Invalid_flag;
int data;
int blv count;
int axwt_tot count;
int tandem_index;
int group_index;
int test;
int axle total;
int axleindex;
int axindex;
int sensor;

```

```

int smpltine;
int final_cnt;
int reset_time;
int reset_ref;
int axle_cnt;
int Request;
int truck_rec;
int two_ft_sensor;
int ten_ft_sensor;
int count;
int previous_sensor_count;
int reference_time;
int reference_sensor;
int st;
int ac;
int psc;
int rt;
int rs;
int weigh_pad; /* weigh pad element in via_st axwt array */
int wp_smpltine; /* weigh pad sample time index */
int axwt_index; /* index for acquisition of individule */
/* axle weights */
int class; /* identifies class of vehicle */
int veh_class;
int SS; /* reference sensor */
int So; /* offset sensor */
int Id;

long gross_wt;
long steer_wt;

float AS;
float speed;
float tr; /* time difference between reference sensors */
float to; /* time since offset sensor was hit */
float V; /* speed (ft/s) */
float Vmph; /* speed (mph) */
float Ds; /* distance traveled since offset sensor hit */
float as; /* axle spacing */

/***** Function Prototypes *****/

float *SpacingTotals();
long *WeightTotals();
long *WeightViolation();
int Classification();
float AxleSpace();

struct via_st {
    int date2;
    int time2;
    int noaxles;
    long axwt[MAXAXLE];
    long axwt_tot[MAXGROUPS];
    float axsp_tot[MAXGROUPS];
}Recs[MAXRECS];

```

THIS PAGE INTENTIONALLY BLANK


```

/*****
irq_din.h
*****/

```

```

/** REGISTERS LOCATION FOR PB-DIN (Piggyback A) OF VMOD 1 **/

```

```

/**** -Timer Registers ****/

```

```

#define TCR1  *(unsigned char*)(0x87FE2421)
#define CPRH1 *(unsigned char*)(0x87FE2427)
#define CPRH1 *(unsigned char*)(0x87FE2429)
#define CPRL1 *(unsigned char*)(0x87FE242B)
#define TCRH1 *(unsigned char*)(0x87FE242F)
#define TCRM1 *(unsigned char*)(0x87FE2431)
#define TCRL1 *(unsigned char*)(0x87FE2433)
#define TSR1  *(unsigned char*)(0x87FE2435)

```

```

/** REGISTER LOCATION FOR PB-DIN (Piggyback A) OF VMOD 2 **/

```

```

/**** Timer Registers ****/

```

```

#define TCR2  *(unsigned char*)(0x87FE4421)
#define CPRH2 *(unsigned char*)(0x87FE4427)
#define CPRH2 *(unsigned char*)(0x87FE4429)
#define CPRL2 *(unsigned char*)(0x87FE442B)
#define TCRH2 *(unsigned char*)(0x87FE442F)
#define TCRM2 *(unsigned char*)(0x87FE4431)
#define TCRL2 *(unsigned char*)(0x87FE4433)
#define TSR2  *(unsigned char*)(0x87FE4435)

```

```

/** REGISTERS LOCATION FOR PB-DIN (Piggyback B) OF VMOD 1 **/

```

```

/**** Timer Registers ****/

```

```

#define TCR3  *(unsigned char*)(0x87FE24A1)
#define CPRH3 *(unsigned char*)(0x87FE24A7)
#define CPRH3 *(unsigned char*)(0x87FE24A9)
#define CPRL3 *(unsigned char*)(0x87FE24AB)
#define TCRH3 *(unsigned char*)(0x87FE24AF)
#define TCRM3 *(unsigned char*)(0x87FE24B1)
#define TCRL3 *(unsigned char*)(0x87FE24B3)
#define TSR3  *(unsigned char*)(0x87FE24B5)

```

```

/** REGISTERS LOCATION FOR PB-DIN (Piggyback B) OF VMOD 2 **/

```

```

/**** Timer Registers ****/

```

```

#define TCR4  *(unsigned char*)(0x87FE44A1)
#define CPRH4 *(unsigned char*)(0x87FE44A7)
#define CPRH4 *(unsigned char*)(0x87FE44A9)
#define CPRL4 *(unsigned char*)(0x87FE44AB)
#define TCRH4 *(unsigned char*)(0x87FE44AF)
#define TCRM4 *(unsigned char*)(0x87FE44B1)
#define TCRL4 *(unsigned char*)(0x87FE44B3)
#define TSR4  *(unsigned char*)(0x87FE44B5)

```

```

/*****

```

```

#define BUSY      1
#define AVAIL     0
#define IDLE     0
#define TIMER1    1
#define TIMER2    2
#define MAX_AX_SENSOR 11
#define True      1
#define False     0
/*****
      FUNCTIONS DECLARATION
*****/

int    Activate_Sensors(),
        Close_Devices(),
        DeActivate_Sensors(),
        delay(),
        Get_Ax_Spacings(),
        Init_Var(),
        Init_S7_S10(),
        Open_Devices(),
        Read_Weight(),
        Reset_VDAD(),
        Service_Sensor1_5(),
        Service_Sensor6_10(),
        VDAD_Login(),
        VDAD_Logout();
/*****
int    Count[MAX_AX_SENSOR], /* Array stores the count per axle sensor */
        TOBS[MAX_AX_SENSOR], /* Array tells which sensor uses which timer */

        Tiner1_Status,
        Timer2_Status,
        Time1, Time2,
        dist, EndRec,
        Endtruck_Flag,
        Axle_Dist_Cnt,
        Total_Ax_Count,
        Starttruck, VehId,
        Temp_Count_S1,
        Temp_Count_S3,
        truck_gone,
        Prev_Total_Ax_Count,
        V1a_sig_H1, V1a_sig_H2,
        V1a_sig_H3, V1a_sig_H4,
        V2a_sig_H1, V2a_sig_H2,
        V2a_sig_H3, V2a_sig_H4,
        V1b_sig_H1, V1b_sig_H2,
        V1b_sig_H3, V1b_sig_H4,
        V2b_sig_H1, V2b_sig_H3,
        din_num1, din_num2,
        din_num3, din_num4,
        din_num5, din_num6,
        din_num7, din_num8,
        pdad1, dev_stat;

        i, sig_in;

FILE    *fp1, *fp3;

```

```
char *din_nam1[20], *din_nam2[20],  
      *din_nam3[20], *din_nam4[20],  
      *din_nam5[20], *din_nam6[20],  
      *din_nam7[20], *din_nam8[20],  
      *dadlin[20];
```

```
control;
```

THIS PAGE INTENTIONALLY BLANK

```

*****
Makefile for IRQ_DIN program
*****

ODIR   = CMDS      # directory for files with no suffix
RDIR   = .../RELS  # relocatable files
SDIR   = .         # source directory (default .)
R       = r68
RFLAGS =
LFLAGS =
CFLAGS =

DEFS    = ..../DEFS
SLIB     = ..../LIB/sys.l
DIXLIB   = /dd/BSP/LIB/pbdix.l
VDADLIB  = /dd/BSP/LIB/vdad.l

SYSDEFS = ${DEFS}/oskdefs.d

irq_din: irq_din.r compute.r
cc irq_din.c compute.c -fd=${ODIR}/${*} -l=${DIXLIB} -l=${VDADLIB} ${CFLAGS} -i
attr ${ODIR}/${*} -e

load:
load /dd/OS9SYS/OBJS/vbf -d
load /dd/BSP/VDAD/OBJS/vbVDADi -d
load /dd/BSP/VDAD/OBJS/dadl -d
load ${ODIR}/irq_din -d

#+++++++ end of makefile ++++++

```

THIS PAGE INTENTIONALLY BLANK

i

```
chd /h0/os9sys/objs
load -d vbf
chd /h0/bsp/vdad/objs
load -d vbVDADi vbVDADo dadlin dad2in dad1 dad2 dad3 dad4 dad5 dad6 dad7 dad8
chd /h0/bsp/vmod/objs
load -d dixVH00 dix0a dix0b dix1a dix1b dix2a dix2b dix3a dix3b
iniz dix0a dix0b dix1a dix1b dix2a dix2b dix3a dix3b
iniz dadlin dad1 dad2 dad3 dad4 dad5 dad6 dad7 dad8
chd /h0/bsp/ran/objs
load -d ran r0_1024k
iniz r0
free r0
```

THIS PAGE INTENTIONALLY BLANK


```
*****
makefile for OS-9/68020 V2.4 VX20 ROMable System
```

```
>>>> type "make -sb" for more information <<<<
*****
```

using:

```
load echo
echo "*****"
echo "makefile for OS-9/68020 V2.4 VX20 ROMable System"
echo
echo
echo "IMPORTANT: Conditional assembly notes:"
echo
echo "The following conditional assembly option is available:"
echo
echo "Flag      Usage"
echo "-----"
echo "COPY_ROM  extends RAM area in memory search list to "
echo "           copy ROM-code into RAM."
echo "           Bootstrap loader "boot_cp.r" must be linked"
echo "           to invoke copy procedure."
echo
echo "A minimal memory search list can be found in "ROM/DEFS/vx20.d"."
echo "If a customized version is needed, the memory search list must"
echo "be changed there and/or in "BSP/VX20/initVX20.a"."
echo
echo "The START macro can be entered to overwrite the default"
echo "value "START=40000000" if a different base address is desired"
echo "for test purposals."
echo
echo
echo "      display the help message"
echo "      -----"
echo "make [using] -sb"
echo
echo
echo "      system ROM modules"
echo "      -----"
echo "without any debugger:"
echo "    keVX20 <addr> [START=<addr>]"
echo "with DEBUG:"
echo "    dbVX20 <addr> [START=<addr>]"
echo "with ROMBUG:"
echo "    rbVX20 <addr> [START=<addr>]"
echo
echo
echo "example:  make keVX20_40000000 -b"
echo "          make dbVX20_40000000 START=40000000 -b"
echo "          make rbVX20_87400000 START=87400000 -b"
echo
echo
echo "      ROMable Systems (examples)"
echo "      -----"
echo "ke_VX20   romable system without any debugger"
echo "db_VX20   romable system with DEBUG"
echo "rb_VX20   romable system with ROMBUG"
echo "tk_VX20   target kit system with DEBUG"
echo
echo
echo "example:  make ke_VX20 -b"
```

```

echo "          make rb_VM20 START=87400000 -b"
echo
echo "*****"

```

```

#####

```

```

START = 40000000 / ROM base address

```

```

ODIR = BOBJS
SDIR = .
SCOM = ../COMMON
RDIR = RELS
ROMLIB = ../LIB

```

```

SYSDEFS = ../DEFS/oskdefs.d ../BSP/DEFS/systype.d
ROMDEFS = ../DEFS/va20.d ../DEFS/cpudefs.d

```

```

CMDS = ../CMDS
CMDSPEP = ../CMDS_PEP

```

```

WIM = ../APPLIC/PB/DIX/CMDS
WIMI = ../WIM

```

```

OBJSSYS = ../OS9SYS/OBJS
OBJSCOM = ../BSP/COMMON/OBJS
OBJSRAM = ../BSP/RAM/OBJS
OBJSV20 = ../BSP/VM20/OBJS
OBJSVSCSI = ../BSP/VSCSI/OBJS
OBJSGO = ../SYSGO/OBJS

```

```

SLIB = ../LIB/sys.l
CLIB = ../LIB/clib.l
CLIBN = ../LIB/clibn.l
MLIB = ../LIB/math.l

```

```

DBGULIB = -l=$(CLIBN) -l=$(MLIB) -l=$(SLIB)
RBUGLIB = -l=$(CLIBN) -l=$(MLIB) -l=$(ROMLIB)/flshcach.l -l=$(SLIB)

```

```

DB0 = $(ROMLIB)/debug.l
DB1 = $(ROMLIB)/dbgdown.l
DB2 = $(ROMLIB)/dbgconn.l
DISL = $(ROMLIB)/disasn.l
RB = $(ROMLIB)/rombug.l

```

```

R = r68

```

```

RFLAGSke = -q
RFLAGSdb = -q -aDEBUGGER
RFLAGSrb = -q -aROMBUG

```

```

LFLAGSke = -swa >=$(ODIR)/map.ke
LFLAGSdb = -swa >=$(ODIR)/map.db
LFLAGSrb = -swa -aj -M=3k >=$(ODIR)/map.rb

```

```

keVM20_$(START):
$(R) $(SCOM)/vectors.a $(RFLAGSke) -o=$(RDIR)/vectors.r

```

```

$(R) $(SCON)/const.a $(RFLAGSke) -o=$(RDIR)/const.r
$(R) $(SCON)/sysboot.a $(RFLAGSke) -o=$(RDIR)/sysboot.r
$(R) sysinit.a $(RFLAGSke) -o=$(RDIR)/sysinit.r
$(R) $(SCON)/io8530.a $(RFLAGSke) -o=$(RDIR)/io8530.r
$(R) $(SCON)/rtc8571.a $(RFLAGSke) -o=$(RDIR)/rtc8571.r
168 $(LFLAGSke) -l=$(SLIB) \
    $(RDIR)/vectors.r $(RDIR)/const.r $(RDIR)/boot_ke.r \
    $(RDIR)/sysboot.r $(RDIR)/sysinit.r \
    $(RDIR)/io8530.r $(RDIR)/rtc8571.r \
    -O=$(ODIR)/$* -r=$(START)

```

dbVM20_\$(START):

```

$(R) $(SCON)/vectors.a $(RFLAGSdb) -o=$(RDIR)/vectors.r
$(R) $(SCON)/const.a $(RFLAGSdb) -o=$(RDIR)/const.r
$(R) $(SCON)/sysboot.a $(RFLAGSdb) -o=$(RDIR)/sysboot.r
$(R) sysinit.a $(RFLAGSdb) -o=$(RDIR)/sysinit.r
$(R) $(SCON)/io8530.a $(RFLAGSdb) -o=$(RDIR)/io8530.r
$(R) $(SCON)/rtc8571.a $(RFLAGSke) -o=$(RDIR)/rtc8571.r
168 $(LFLAGSdb) $(DBGULIB) \
    $(RDIR)/vectors.r $(RDIR)/const.r $(RDIR)/boot_db.r \
    $(RDIR)/sysboot.r $(RDIR)/sysinit.r \
    $(RDIR)/io8530.r $(RDIR)/rtc8571.r \
    $(DBG) $(DISL) -O=$(ODIR)/$* -r=$(START)

```

rbVM20_\$(START):

```

$(R) $(SCON)/vectors.a $(RFLAGSrb) -o=$(RDIR)/vectors.r
$(R) $(SCON)/const.a $(RFLAGSrb) -o=$(RDIR)/const.r
$(R) $(SCON)/sysboot.a $(RFLAGSrb) -o=$(RDIR)/sysboot.r
$(R) sysinit.a $(RFLAGSrb) -o=$(RDIR)/sysinit.r
$(R) $(SCON)/io8530.a $(RFLAGSrb) -o=$(RDIR)/io8530.r
$(R) $(SCON)/rtc8571.a $(RFLAGSrb) -o=$(RDIR)/rtc8571.r
168 $(LFLAGSrb) $(RBUGLIB) \
    $(RDIR)/vectors.r $(RDIR)/const.r $(RDIR)/boot_rb.r \
    $(RDIR)/sysboot.r $(RDIR)/sysinit.r \
    $(RDIR)/io8530.r $(RDIR)/rtc8571.r \
    $(RB) -O=$(ODIR)/$* -r=$(START)

```

```

# 4 x 27512 EPROMs = 256 KByte
# - no debugger included
# - rbf is included
# - sysgo module is sysgo

```

ke_VM20:

```

merge $(ODIR)/keVM20_$(START) \
    $(OBJSSYS)/kern020 $(OBJSSYS)/scach020 \
    $(OBJSSYS)/rbf $(OBJSSYS)/scf $(OBJSSYS)/vbf \
    $(OBJSSYS)/pipexan $(OBJSSYS)/pipe $(OBJSSYS)/null $(OBJSSYS)/nil \
    $(OBJSVH20)/initVM20 $(OBJSVH20)/clkVM20 \
    $(OBJSVH20)/scvVM20 $(OBJSVH20)/tvn20_0 $(OBJSVH20)/tvn20_1 \
    $(OBJJSRAM)/raa $(OBJJSRAM)/r0_4096k \
    $(OBJSGO)/startgo \
    $(OBJSGO)/sysgo \
    >=$(ODIR)/ke_VM20
merge $(CMDS)/shell $(CMDS)/cio020 $(CMDS)/procs $(CMDS)/math \
    $(CMDS)/date $(CMDS)/mdir $(CMDS)/free $(CMDS)/iniz $(CMDS)/deiniz \
    $(CMDS)/link $(CMDS)/unlink $(CMDS)/tmode $(CMDS)/tmon \
    $(CMDS)/dir $(CMDS)/cio $(CMDS)/list $(CMDS)/load $(CMDS)/echo \

```

```

>+$(ODIR)/ke_VM20
merge $(WIMI)/dixVHOD $(WIMI)/dix0a $(WIMI)/dix0b $(WIMI)/dix1a \
$(WIMI)/dix1b $(WIMI)/dix2a $(WIMI)/dix2b $(WIMI)/dix3a \
$(WIMI)/dix3b $(WIMI)/vbVDADi $(WIMI)/dadlin $(WIMI)/irq_din \
>+$(ODIR)/ke_VM20
merge $(CMDS)/irqs $(CMDS)/events $(CMDS)/devs \
$(CMDS)/dump $(CMDS)/echo $(CMDS)/help $(CMDS)/ident \
$(CMDS)/printenv $(CMDS)/sleep $(CMDS)/lma \
$(CMDSPEP)/go \
>+$(ODIR)/ke_VM20

```

```

*****

```

```

# 4 x 27512 EPROMs = 256 KByte
# - debug is included
# - rbf is included
# - sysgo module is diskgo
db_VH20:

```

```

merge $(ODIR)/dbVM20 $(START) \
$(OBJSSYS)/kern020 $(OBJSSYS)/scach020 \
$(OBJSSYS)/rbf $(OBJSSYS)/scf \
$(OBJSSYS)/pipeman $(OBJSSYS)/pipe $(OBJSSYS)/null $(OBJSSYS)/nil \
$(OBJSVH20)/initVM20 $(OBJSVH20)/clkVM20 \
$(OBJSVH20)/scvVM20 $(OBJSVH20)/tvn20_0 $(OBJSVH20)/tvn20_1 \
$(OBJSRAM)/ram $(OBJSRAM)/r0_128k \
$(OBJSGO)/diskgo \
>+$(ODIR)/db_VH20
merge $(OBJSCOM)/rbscsi $(OBJSVSCSI)/phVSCSI $(OBJSVSCSI)/c4a2611s.dd \
$(OBJSVSCSI)/dc4_d \
>+$(ODIR)/db_VH20
merge $(CMDS)/shell $(CMDS)/cio020 $(CMDS)/procs $(CMDS)/setine \
$(CMDS)/date $(CMDS)/mdir $(CMDS)/mfree $(CMDS)/iniz $(CMDS)/deiniz \
$(CMDS)/link $(CMDS)/unlink $(CMDS)/tmode $(CMDS)/xmode \
>+$(ODIR)/db_VH20
merge $(CMDS)/tsmon $(CMDS)/irqs $(CMDS)/events $(CMDS)/devs \
$(CMDS)/dump $(CMDS)/echo $(CMDS)/help $(CMDS)/ident \
$(CMDS)/break $(CMDS)/printenv $(CMDS)/sleep \
$(CMDS)/lma \
>+$(ODIR)/db_VH20

```

```

*****

```

```

# 4 x 27512 EPROMs = 256 KByte
# - rombug is included
# - rbf is included
# - sysgo module is sysgo
rb_VH20:

```

```

merge $(ODIR)/rbVM20 $(START) \
$(OBJSSYS)/kern020 $(OBJSSYS)/scach020 \
$(OBJSSYS)/rbf $(OBJSSYS)/scf \
$(OBJSSYS)/pipeman $(OBJSSYS)/pipe $(OBJSSYS)/null $(OBJSSYS)/nil \
$(OBJSVH20)/initVM20 $(OBJSVH20)/clkVM20 \
$(OBJSVH20)/scvVM20 $(OBJSVH20)/tvn20_0 $(OBJSVH20)/tvn20_1 \
$(OBJSRAM)/ram $(OBJSRAM)/r0_128k.dd \
$(OBJSGO)/sysgo \
>+$(ODIR)/rb_VH20
merge $(CMDS)/shell $(CMDS)/cio020 $(CMDS)/procs $(CMDS)/setine \
$(CMDS)/date $(CMDS)/mdir $(CMDS)/mfree $(CMDS)/iniz $(CMDS)/deiniz \
$(CMDS)/link $(CMDS)/unlink $(CMDS)/tmode \

```

```

>+$(ODIR)/rb_VH20
merge $(CHDS)/irqs $(CHDS)/events $(CHDS)/devs \
$(CHDS)/echo $(CHDS)/ident \
$(CHDS)/break $(CHDS)/sleep \
$(CHDS)/lma \
>+$(ODIR)/rb_VH20

```

```

/*****

```

```

# 4 x 27512 EPROMs = 256 KByte

```

```

# - debug included

```

```

# - rbf is included

```

```

# - sysgo module is sysgo

```

```

tk_VH20:

```

```

merge $(ODIR)/dbVH20_$(START) \
$(OBJSSYS)/kern020 $(OBJSSYS)/scach020 \
$(OBJSSYS)/rbf $(OBJSSYS)/scf \
$(OBJSSYS)/pipeman $(OBJSSYS)/pipe $(OBJSSYS)/null $(OBJSSYS)/nil \
$(OBJSVH20)/initVH20 $(OBJSVH20)/clkVH20 \
$(OBJSVH20)/sccVH20 $(OBJSVH20)/tvh20_0 $(OBJSVH20)/tvh20_1 \
$(OBJSRAM)/ram $(OBJSRAM)/ro_128k.dd \
$(OBJSGO)/sysgo \
>+$(ODIR)/tk_VH20

```

```

merge $(CHDS)/shell $(CHDS)/cio020 $(CHDS)/procs $(CHDS)/setine \
$(CHDS)/date $(CHDS)/mdir $(CHDS)/mfree $(CHDS)/iniz $(CHDS)/deiniz \
$(CHDS)/link $(CHDS)/unlink $(CHDS)/tnode $(CHDS)/xnode \
>+$(ODIR)/tk_VH20

```

```

merge $(CHDS)/irqs $(CHDS)/events $(CHDS)/devs $(CHDS)/math \
$(CHDS)/binex $(CHDS)/exbin $(CHDS)/dump $(CHDS)/sleep \
$(CHDS)/echo $(CHDS)/help $(CHDS)/ident $(CHDS)/printenv \
$(CHDS)/kerait $(CHDS)/lma \
$(CHDS)/break \
>+$(ODIR)/tk_VH20

```

```

##### E O P #####

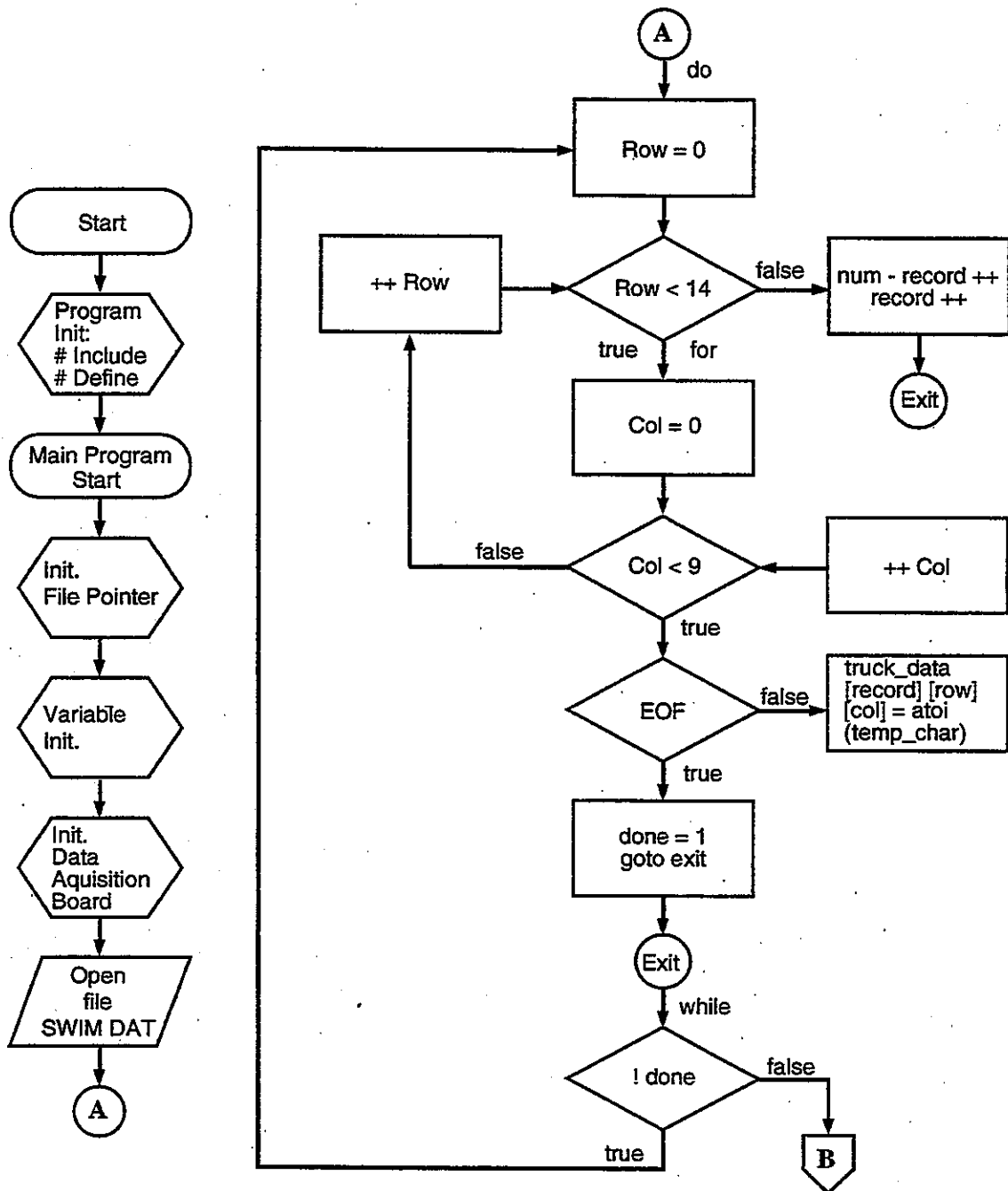
```

THIS PAGE INTENTIONALLY BLANK

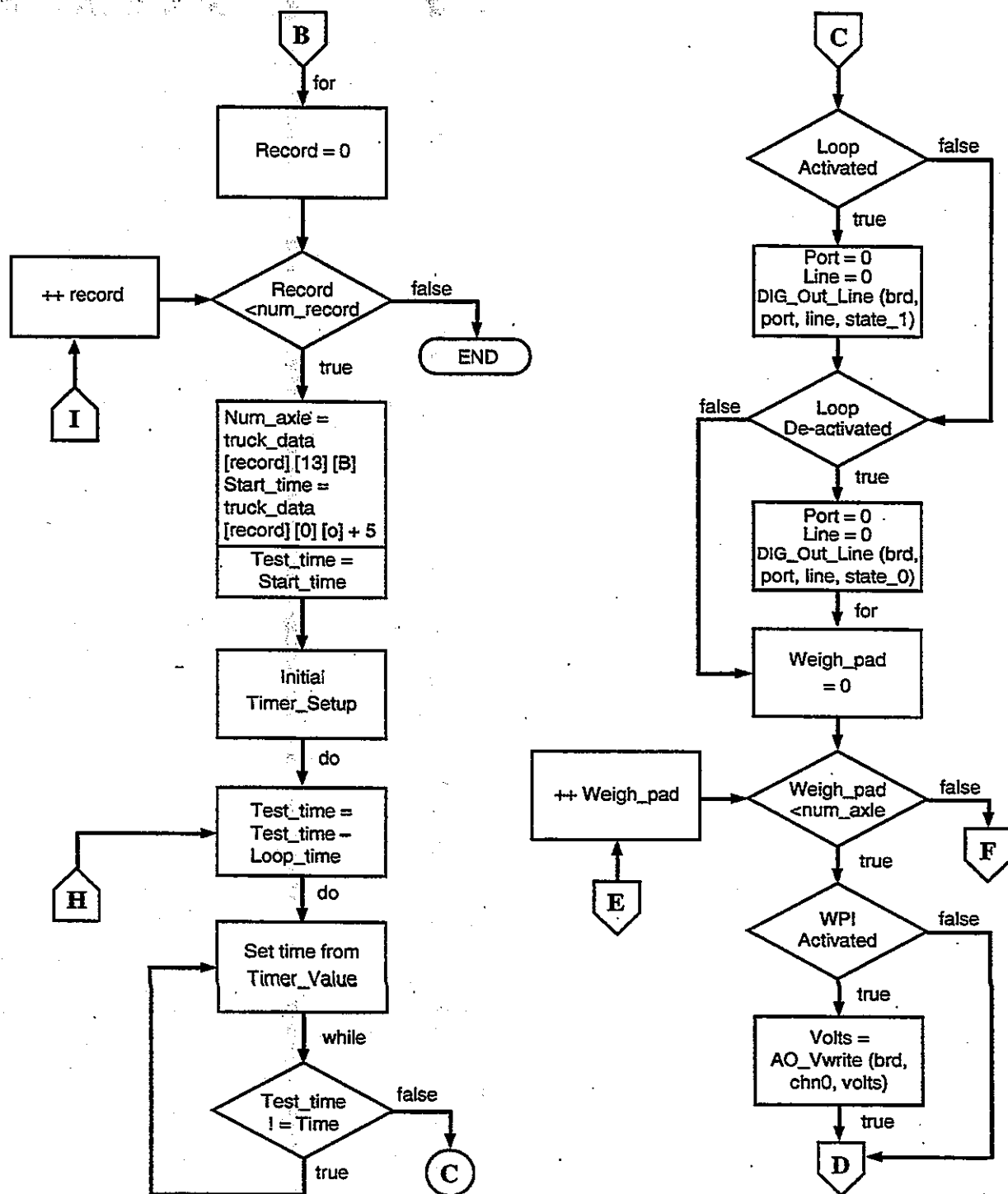
prom_wim

make -b keVH20_400000000 START=400000000
make -b ke_VH20
chd /h0/rom/vm20/boobjs
copy -r ke_VH20 ke
romsplit -q ke

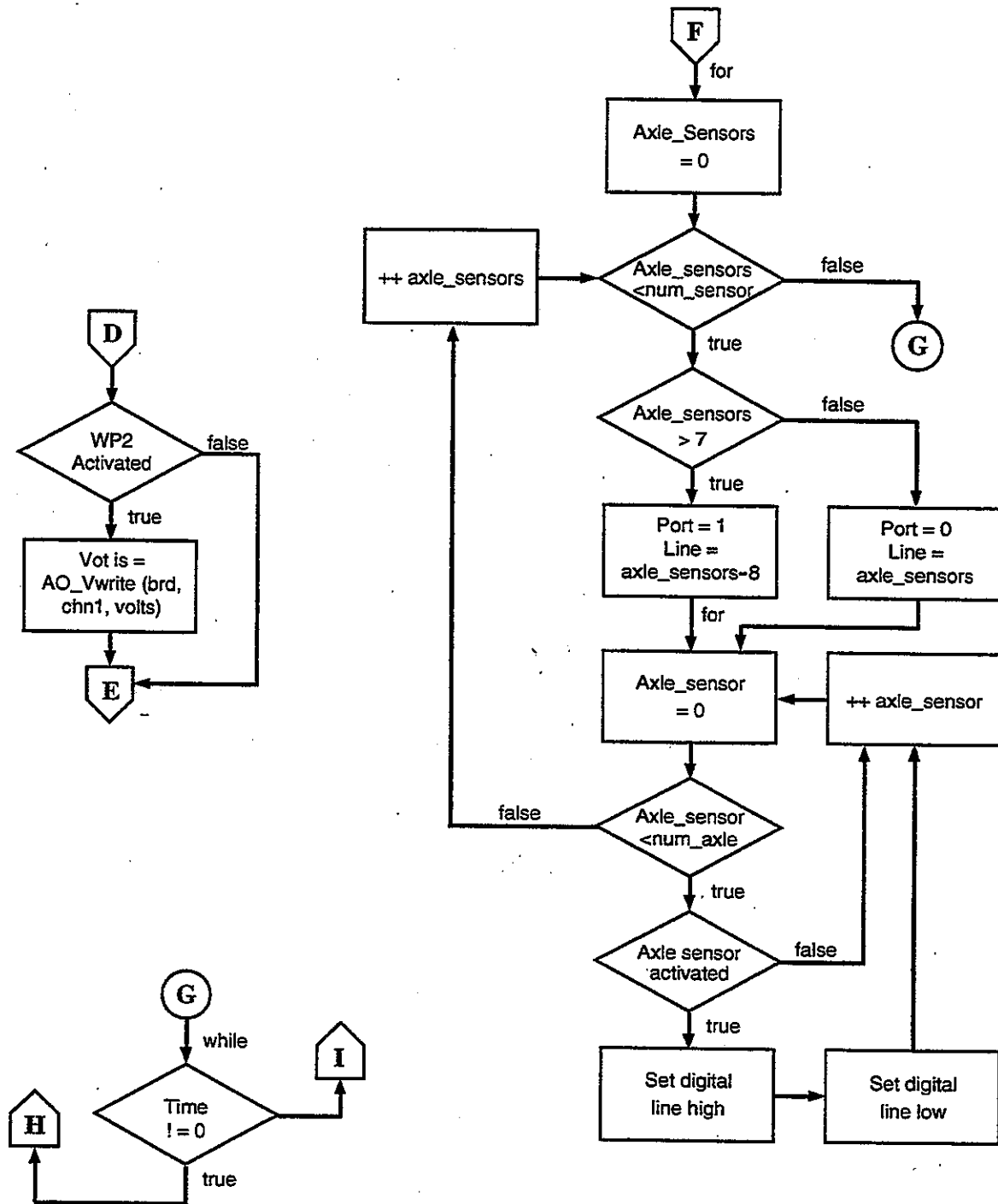
THIS PAGE INTENTIONALLY BLANK



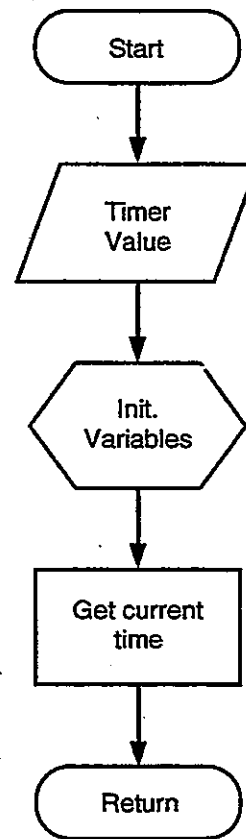
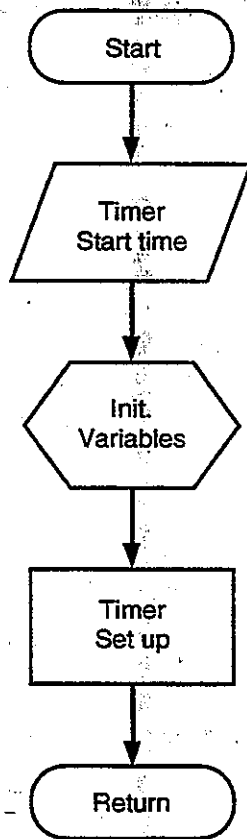
SWIM SIMULATION SOFTWARE FLOW CHART



SWIM SIMULATION SOFTWARE FLOW CHART (con't)



SWIM SIMULATION SOFTWARE FLOW CHART (con't)



SWIM SIMULATION SOFTWARE FLOW CHART (con't)

/******

Truck Simulation For Slow Weigh In Motion (SWIM)

This program simulates a truck traveling through a SWIM system. The SWIM site contains a 6' loop followed by a 2' long weigh scale followed by 11 1' long axle sensors. The truck can vary in number of axles, weight per axles, and speed. The simulation data for each truck is generated using a data simulation program in LOTUS. The program first opens and reads a file SWIM.DAT from drive A and writes the simulation data to a three dimensional array called truck_data. The data from truck_data is then written in real time to the data acquisition board for input to a host computer to process.

Written by Alan Benson on 7/28/92.

*****/

/***** Program Initialization *****/

```
#include <stdio.h> /* Standard I/O */
#include <dos.h>
#include "C:\pclabdrv\pclabdrv.h" /* Function definitions */
#include "C:\pclabdrv\pclabdrv.err" /* Error code definitions */

#define Axles 9 /* Max number of axles on trucks */
#define Sensors 14 /* Number of sensors in system */
#define Trucks 3 /* Max number of trucks for simulation */
#define Swim_data "A:\SWIM.DAT" /* File containing simulation
...data */
#define Block_size 14*9 /* Size of data for 1 truck */
```

/***** Main Program *****/

```
main()
{
```

```
FILE *sdpt; /* Pointer to pre-define structure FILE */
```

/***** Variable Initialization *****/

```
int record = 0, /* Loop variable for trucks */
    row, /* Loop variable for sensors */
    col, /* Loop variable for axles */
```

```

weigh_pad, /* Loop variable for weigh pad */
axle_sensors, /* Loop variable for axle sensors */
axle_sensor, /* Loop variable for 1 axle sensor */
num_axle, /* Max number of axles on trucks */
num_record=0, /* Number of trucks in truck_data */
num_sensor= 12, /* Number of sensors in system */
port, /* Digital port */
chan0 = 0, /* Chan DAC0 output */
chan1 = 1, /* Chan DAC1 output */
mode = 0, /* No handshaking mode */
dir = 1, /* Port is output */
line, /* Port line */
state_0 = 0, /* Digital line state logic (low) */
state_1 = 1, /* Digital line state logic (high) */
truck_data [Trucks][Sensors][Axles], /* 3-D array for
...truck data */

brd = 1, /* Slot number of board */
brd_code, /* Holds the board type */
dump, /* Dummy variable */
done = 0, /* A flag used to determine when to
...exit loop. */
loop_time = 5, /* Time for one loop of main program */
start_time; /* Last time entry in truck_data */

unsigned time; /* Time of free running timer */

float volts; /* Input value form keyboard */

char temp_char[10]; /* Temp storage for char conversion */

time_t tstart, tend;

/***** Initialize Data Aquisition Board

Configure DA board to initial conditions
and digital port A and B for output. *****/

Get_DA_Brds_Info (brd, &brd_code, &dump, &dump, &dump,
&dump, &dump, &dump, &dump);

port = 1;
DIG_Prt_Config (brd, port, mode, dir);
port = 0;
DIG_Prt_Config (brd, port, mode, dir);

/***** Open and Read file SWIM.DAT from drive A

```

and write data to 3-D array truck_data.
 The data in SWIM.DAT is stored as text
 so the command "atoi" is used to convert
 from test to an integer before storing
 in truck_data. *****/

```
sdpt = fopen(Swim_data, "rt");       /* Open file SWIM.DAT */
do
{
    for (row=0; row<14; ++row)   /* Loop for rows */
    {
        for (col=0; col<9; ++col) /* Loop for columns */
        {
            /* If End Of File (EOF) exit to while loop */

            if (fscanf(sdpt, "%s", temp_char) == EOF)
            {
                done = 1; /* Set flag to 1 */
                goto exit; /* Exit to while loop */
            }
            /* Convert temp_char from text to
            interger and store in truck_data */

            truck_data[record][row][col]
                = atoi(temp_char);
        }
    }
    num_record++; /* Number of trucks in system */
    record++;     /* Truck loop for truck_data */
exit: ;
}
while (!done);     /* If done = 1 then exit while loop */
fclose(sdpt);     /* Close file SWIM.DAT */

/***** Main Program Loop
```

The main loop outputs data, stored in truck_data,
 to the Data Aquisition (DA) board in accordance
 with the system configuration. The system is
 configured starting with a 6' loop followed by a
 2' long weigh scale followed by 11 1' long axle
 sensors. The loop output from the DA board is a
 digital signal that is set high when the truck
 enters the system and is set low when the truck
 leaves the system. The weigh pad output from

the DA board in an analog signal which is equal to 1V = 10,000lb. The axle sensor output from the DA board is a digital signal that is set high when the truck axle first hits it and is set low after a certain time response delay which is calculated from the speed of the vehicle. The main loop can handel multiple trucks through the system depending upon how many are stored in truck_data. *****/

```
for (record=0; record<num_record; ++record)
{
    num_axle = truck_data[record][13][0];
    start_time = truck_data[record][0][0] + 5;
    test_time = start_time;
    Timer_Setup(start_time);
    do
    {
        test_time = test_time - loop_time;
        do
        {
            time = Timer_Value();
        }
        while (test_time != time);

        /***** - Loop -

            Loop is set high when the
            truck first enters the loop
            and is set low when the truck
            leaves the loop. *****/

        /***** Set digital line A0 high *****/

        if (truck_data[record][0][0] >= test_time &&
            truck_data[record][0][0] < test_time + loop_time)
        {
            port = 0;
            line = 0;
            DIG_Out_Line (brd, port, line, state_1);
        }

        /***** Set digital line A0 low *****/

        if (truck_data[record][2][num_axle - 1] >= test_time &&
```



```

truck_data[record][2][num_axle - 1] < test_time
+ loop_time)
{
    port = 0;
    line = 0;
    DIG_Out_Line (brd, port, line, state_0);
}

/***** - Weigh Pads 1 and 2 -

As each axle crosses the weigh
pad an analog voltage is outputed
on analog channel 0 for WP1 and
channel 1 for WP2.

1V = 10,000lbs          *****/

for (weigh_pad=0; weigh_pad<num_axle; ++weigh_pad)
{

    /***** Output voltage on channel 0 *****/

    if (truck_data[record][2][weigh_pad] >= test_time
        && truck_data[record][2][weigh_pad] <
        test_time + loop_time)
    {
        volts = (truck_data[record][12][weigh_pad]/10.0);
        AO_VWrite( brd, chan0, volts);
    }

    /***** Output voltage on channel 1 *****/

    if (truck_data[record][4][weigh_pad] >= test_time
        && truck_data[record][4][weigh_pad] <
        test_time + loop_time)
    {
        volts = (truck_data[record][12][weigh_pad]/10.0);
        AO_VWrite( brd, chan1, volts);
    }
}

/***** - Axle Sensors -

As each axle crosses an axle

```

sensor a digital voltage is
outputted on the specified
digital line.

Axle Sensor	Port Line
S0	A3
S1	A4
S2	A5
S3	A6
S4	A7
S5	B0
S6	B1
S7	B2
S8	B3
S9	B4
S10	B5

*****/

```
for (axle_sensors=1; axle_sensors<num_sensor;
    ++axle_sensors)
```

```
{
```

```
    /***** Check for port A or B
            and set line number *****/
```

```
    if (axle_sensors > 7)
    {
        port = 1;
        line = axle_sensors - 8;
    }
    else
    {
        port = 0;
        line = axle_sensors;
    }
}
```

```
for (axle_sensor=0; axle_sensor<num_axle;
    ++axle_sensor)
```

```
{
```

```
    /***** Set axle sensor port
            line high if equal to
            test_time *****/
```

```
    if (truck_data[record][axle_sensors][axle_sensor]
```

```

        >= test_time && truck_data[record][axle_sensors]
        [axle_sensor] < test_time + loop_time)
    {
        DIG_Out_Line (brd, port, line, state_1);

        /***** Set axle sensor digital
                line low          *****/

        DIG_Out_Line (brd, port, line, state_0);
    }
}
    } while (time != 0);
}
}

```

**** - Timer_Setup -

This subroutine sets up timer B1 on the data
aquisition board for 16-bit binary timing.
The main program pass start_time for starting
the down timer. *****/

Timer_Setup(int start_time)

```

{
    int brd = 1, /* Slot number of board */
        ctr = 1, /* Counter B1 */
        mode = 0, /* Toggle output from low to high */
        bin = 1, /* 16-bit binary counter */

    ICTR_Setup (brd, ctr, mode, start_time, bin);
}

```

**** - Timer_Value -

This subroutine reads timer B1 and stores
the 16-bit binary number in "time". The
subroutine then passes "time" back to the
main program. *****/

Timer_Value()

```

{

```

```
int brd = 1, /* Slot number of board */  
    ctr = 1, /* Timer B1 */
```

```
unsigned time; /* Current timer value */
```

```
ICTR_Read (brd, ctr, &time);  
return(time);
```

```
}
```

APPENDIX E

Bibliography

- 1) "IEEE Standard for a Versatile Backplane Bus: VMEbus (ANSI/IEEE Std 1014-1987), "The Institute of Electrical and Electronics Engineers, Inc., Wiley-Interscience Inc., New York, 1988.
- 2) "Software for Advanced Traffic Controllers," Bullock, D., Hendrickson, C., Technical Report, Louisiana State University, Carnegie Mellon University, November 1992.
- 3) "C Primer plus," Waite, M., Prata, S., Martin, D., Howard W. Sams & Co., Indianapolis, IN, 1986.
- 4) "The OS-9 Catalog," Microware Systems Corporation, Des Moines, Iowa, 1992.
- 5) "Standard specification for Highway Weigh-In-Motion (WIM) Systems with user Requirements and Test Method," American Society for Testing and Materials, Philadelphia, PA, February, 1992.
- 6) "Special Provisions for Construction on State Highways," California Department of Transportation, Sacramento, CA.
- 7) "Evaluation of the PAT and Streeteramet Weigh-In-Motion Systems," Chow, W., California Department of Transportation, Sacramento, CA, 1982.
- 8) "Evaluation of Computer Hardware and High-Level Language Software for Field Traffic Control," Quinlan, T., California Department of Transportation, Sacramento, CA, 1989.
- 9) PEP Modular Computers Inc., Technical Product Reports, Pittsburgh, PA.
- 10) International Road Dynamics Inc., Technical Product Reports, Saskatchewan, Canada.
- 11) PAT Equipment Corporation, Technical Product Reports, Chambersburg, PA.
- 12) Detector Systems, Technical Product Report, Stanton, CA.

